Using Gelato Web3 Function on Berachain

See these full GitHub Project Code Repositories.

https://github.com/gelatodigital/how-tos-2-w3f-speed-run

https://github.com/gelatodigital/how-tos-3-w3f-triggers

This developer guide will walk you through setting up a new Web3 Function app and achieve automation for your contract, configuring the Berachain network details, setup basic wallet connection, test and deploy your contract through a hardhat configuration.

What are Gelato Web3 Functions ?

Gelato's Web3 Functions are a powerful automation system designed to enhance Web3 operations. Web3 Functions serve as a comprehensive tool, enabling developers to effortlessly set up, manage, and automate their smart contract tasks.

Use Case Examples

- **Oracles**: create fine-tuned customized oracles pushing prices on-chain following predefined logic (Pyth Oracle Poc , RedStone Oracle Poc)
- **Real-time market data:** Ostium uses high-throughput automation for real-time market data integration, precise order execution, and efficient liquidation management.
- **Airdrop Claiming:** Automate airdrop claiming with configurable plans stored on-chain (Github repository)
- **PancakeSwap:** Native Limit Orders
- **Content creation:** Automated content creation on Lens using ChatGPT

How do Gelato Web3 functions work?

Gelato Web3 Functions offer a flexible and efficient way to automate tasks. Depending on your needs, you can choose between off-chain and on-chain computations. You can trigger functions based on time, events, or every block, and execute custom logic using TypeScript for off-chain or Solidity for on-chain processes. Once a Web3 function is created, it manages automated transactions or smart contract interactions, with built-in monitoring to ensure proper execution and allow for real-time adjustments.

Implementation path

Step	Description

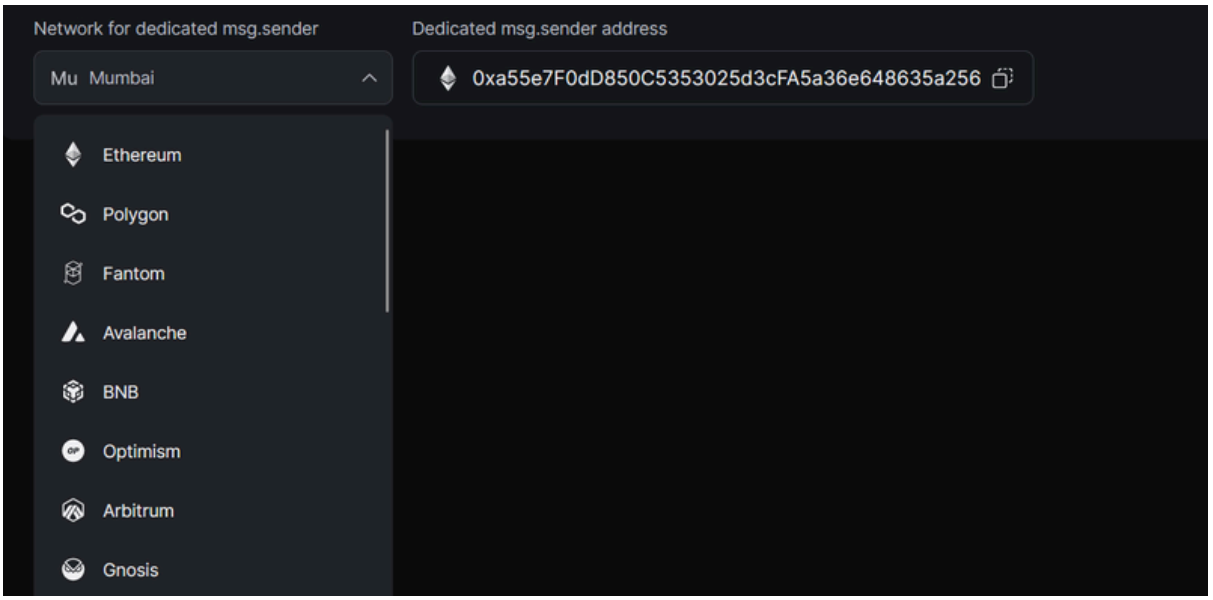
How do you want to trigger your web3 function?	Start by deciding on the type of trigger you want to use. (Time, event, or every block)
What to run?	- Typescript Function (custom logic, off-chain computation) - Solidity Function (on-chain computation, custom logic) - Transaction
Task Creation	Create a Web3 Function task to allow the execution of typescript, solidity or transaction
Finalize & Monitor	Once you've defined your function ensure you monitor its execution to confirm that it works as expected. Make any necessary adjustments.

Ways to submit Web3 Function Tasks

- **Using UI:** This method is ideal for developers or operators who prefer a graphical interface for task configuration. For instance, if you're a developer who wants to quickly test a function or a non-technical person looking to schedule a recurring job without writing any code.
- **Using Smart contract:** You can create a task that uses Web3 Function from your smart contract as well. If your project involves complex interactions and you need the task creation to be a part of an on-chain transaction, you would directly interact with a smart contract.
- **Using Automate SDK:** The SDK is suitable when you need to integrate task creation into your development environment or automated scripts. It's also useful for complex setups that require conditional logic before task submission.

For security reasons, during task creation, you will see an address which will be the `msg.sender` for your task executions.
If you are the owner of the target contract in question, it's recommended to implement a `msg.sender` restriction within your smart contract. This involves whitelisting a dedicated `msg.sender` address. Such a measure ensures that only tasks you have created can call your function, significantly elevating the security posture of your operations. For a hands-on

guide and to manage your dedicated `msg.sender` settings, please connect to the app and visit your own Settings page.



How to pay

- **Transaction pays for itself**: You can choose to have your function pay the fee during executions. It must be remembered that running Web3 Functions has computational costs. Please see here the Free Tier limits
- **1Balance:** in the case that the Web3 Functions goes above these limits, 1Balance will be also required to pay for the computational costs. For example, a user can top up their Gelato 1Balance using USDC on Polygon. This USDC balance will now be used to cover all gas costs and fees

In this guide, we'll use the UI for submitting a Web3 function.

Steps to Set Up

Requirement

Before beginning, make sure you have the following installed or set up on your computer beforehand.

- NVM or Node v18.18.2
- npm, yarn, or pnpm
- Wallet that contains BERA token (for deployment)

How to Write a Web3 Function to Update Price Using the CoinGecko API

In this guide we will jump through creating a Web3 Function in TypeScript that fetches the latest price of Ethereum from the CoinGecko API and updates an oracle smart contract with this data.
Follow these steps to set up and deploy your Web3 Function.

Set Up the Development Environment

1. **Clone the Template Repository:**
Begin by cloning the Web3 Function Hardhat template:

```
JavaScript
git clone
https://github.com/gelatodigital/web3-functions-hardhat-template.git
cd web3-functions-hardhat-template
```

2. **Install Dependencies:**
Install the necessary dependencies:

```
JavaScript
yarn install
```

3. **Configure Environment Variables:**
Copy the example environment file and fill in your Alchemy API key for local testing:

```
None
cp .env.example .env
```

Update the `.env` file with your details:

```
None
ALCHEMY_ID=your_alchemy_key
PRIVATE_KEY=your_private_key  # Optional, needed for contract deployment
```

Write the Web3 Function

1. **Create the Function:**

Below is an example of a TypeScript function that interacts with the CoinGecko API and updates the oracle contract. Here's a simplified example:

JavaScript

```
import { Web3Function, Web3FunctionContext } from
"@gelatonetwork/web3-functions-sdk";
   import { Contract, ethers } from "ethers";
   import ky from "ky";

   const ORACLE_ABI = [
     "function lastUpdated() external view returns(uint256)",
     "function updatePrice(uint256)",
   ];

   Web3Function.onRun(async (context: Web3FunctionContext) => {
     const { multiChainProvider } = context;
     const provider = multiChainProvider.default();

     // Define Oracle contract address and instance
     const oracleAddress = context.userArgs.oracle ||
"0x71b9b0f6c999cbbb0fef9c92b80d54e4973214da";
     const oracle = new Contract(oracleAddress, ORACLE_ABI, provider);

     // Get the last update timestamp from the oracle
     const lastUpdated = parseInt(await oracle.lastUpdated());
     const nextUpdateTime = lastUpdated + 300; // Update every 5 minutes

     // Check if it's time to update the price
     const currentTimestamp = (await provider.getBlock("latest")).timestamp;
     if (currentTimestamp < nextUpdateTime) {
       return { canExec: false, message: "Time not elapsed" };
     }

     // Fetch the latest price from CoinGecko
     const currency = context.userArgs.currency || "ethereum";
     const priceData = await ky.get(

`https://api.coingecko.com/api/v3/simple/price?ids=${currency}&vs_currencies
=usd`,
       { timeout: 5000, retry: 0 }
     ).json();
     const price = Math.floor(priceData[currency].usd);

     // Update the oracle with the new price
     return {
       canExec: true,
       callData: [{
         to: oracleAddress,
```

```
        data: oracle.interface.encodeFunctionData("updatePrice", [price])
      }],
    };
  });
```

2. **Configure Runtime Settings:**
   Create a `schema.json` file to define the runtime configuration:

```JavaScript
  {
    "web3FunctionVersion": "2.0.0",
    "runtime": "js-1.0",
    "memory": 128,
    "timeout": 30,
    "userArgs": {
      "currency": "string",
      "oracle": "string"
    }
  }
```

3. **Provide User Arguments:**
   Create a `userArgs.json` file to test the function locally:

```JavaScript
{
  "currency": "ethereum",
  "oracle": "0x71B9B0F6C999CBbB0FeF9c92B80D54e4973214da"
}
```

3. Test and Deploy the Function

1. **Run Locally:**
   Test the Web3 Function locally to ensure it works as expected:

```None
npx hardhat w3f-run oracle --logs
```

If you want to run tests locally

```
None
npx hardhat test web3-functions/simple/index.ts --logs --chain-id=31337
```

when testing locally, we can provide the different providers by including them in `.env` at the root folder

```
None

// .env file

PROVIDER_URLS=RPC1,RPC2 //for local testing you can use localhost
```

2. **Deploy the Web3 Function:**
   Once tested, deploy the Web3 Function to IPFS and create an automated task using Gelato's SDK. Before deploying your Web3 Function, it's important to test it locally to ensure everything works as expected.

```
None
npx hardhat w3f-run W3FNAME
```

   Replace `W3FNAME` with the name of your function. For example, if your function is named `oracle`, run:

```
None
npx hardhat w3f-run oracle
```

2. Optional Flags:
   You can use additional flags to get more detailed output:
   - `--logs`: Show internal Web3 Function logs.
   - `--debug`: Display runtime debug messages.

- `--network [NETWORK]`: Set the default runtime network and provider.

Example with flags:

```
None
npx hardhat w3f-run oracle --logs --network hardhat
```

3. Interpreting the Output:
After running your Web3 Function, you will see various outputs, including build results, user argument validations, execution logs, and runtime statistics. These details help you understand the function's behavior and performance.

Example output:

```
None
Web3Function Build result:
   ✓ Schema: /path/to/oracle/schema.json
   ✓ Built file: /path/to/.tmp/index.js
   ✓ File size: 2.47mb
   ✓ Build time: 947.91ms

   Web3Function user args validation:
   ✓ currency: ethereum
   ✓ oracle: 0x71B9B0F6C999CBbB0FeF9c92B80D54e4973214da

   Web3Function running logs:
   > Last oracle update: 0
   > Next oracle update: 3600
   > Updating price: 1898

   Web3Function Result:
   ✓ Return value: {
     canExec: true,
     callData: [
       {
         to: '0x71B9B0F6C999CBbB0FeF9c92B80D54e4973214da',
         data:
'0x8d6cc56d000000000000000000000000000000000000000000000000000000000000076a'
       }
     ]
   }

   Web3Function Runtime stats:
   ✓ Duration: 1.35s
```

```
✓ Memory: 113.55mb
✓ Storage: 0.03kb
✓ Rpc calls: 3
```

Deploying TypeScript Functions

Once you've tested your function locally, the next step is to deploy it to IPFS.

1. Deploy the Web3 Function:
   Use the following command to compile your Web3 Function and upload it to IPFS:

```
None
npx hardhat w3f-deploy W3FNAME
```

   Replace `W3FNAME` with the name of your function. For example:

```
None
npx hardhat w3f-deploy oracle
```

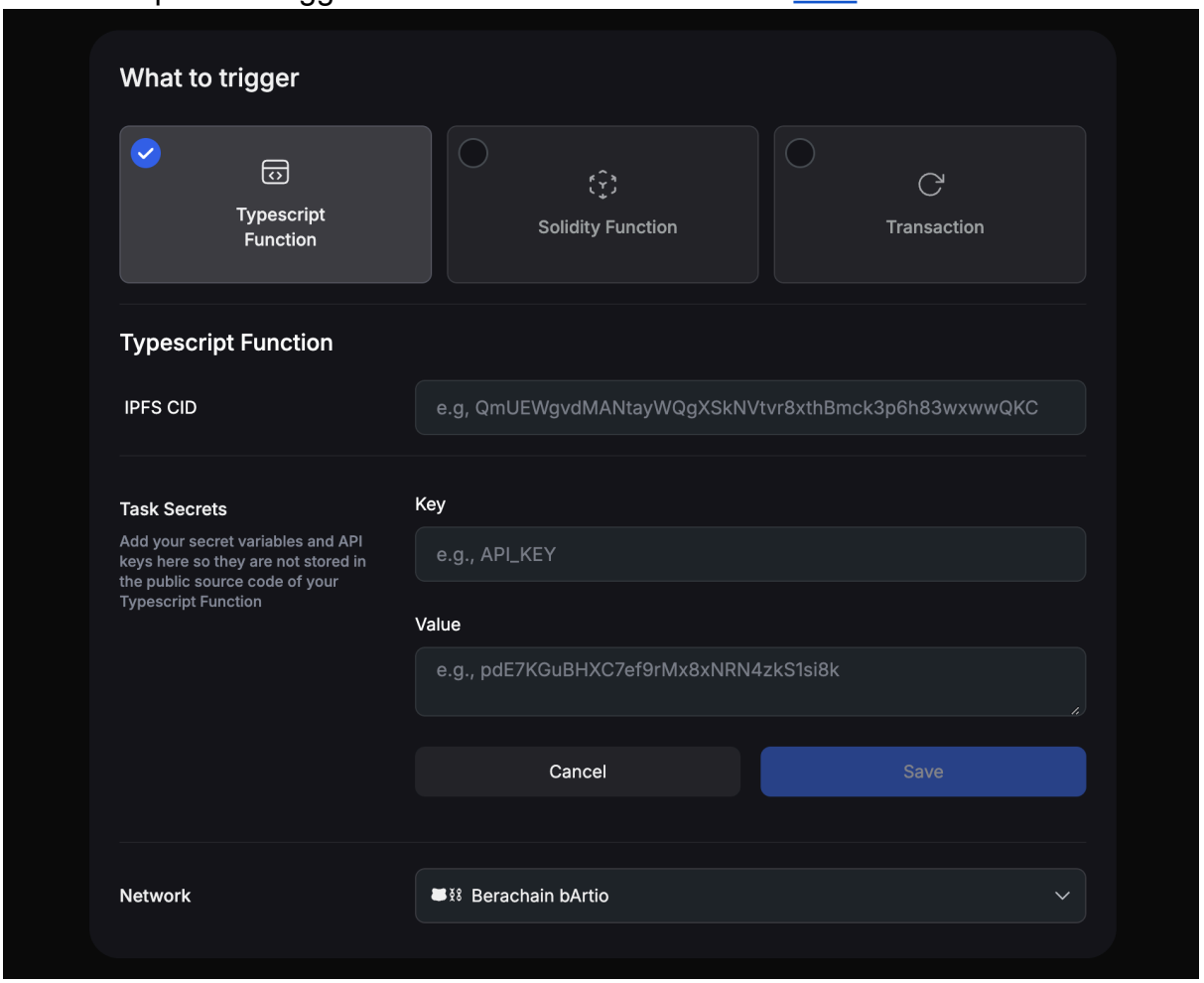
2. Retrieve the IPFS CID:
   After deployment, the IPFS Content Identifier (CID) of your Web3 Function will be returned. This CID is unique to each version of your function and will be needed when creating tasks.

   Example output:

```
None
✓ Web3Function deployed to ipfs.
✓ CID: QmbQJC5XGpQUsAkLq6BqpvLtD8EPNDEaPqyFf4xK3TM6xj
```

Creating a Web3 Function Task using the UI

After deploying your function, you can create a task in Gelato to run it automatically based on specified triggers. You can access this feature here.



1. Selection of Function:
   Go to the "What to trigger" section in the Gelato UI. In the "Typescript Function" subsection, enter the IPFS CID you obtained after deployment.

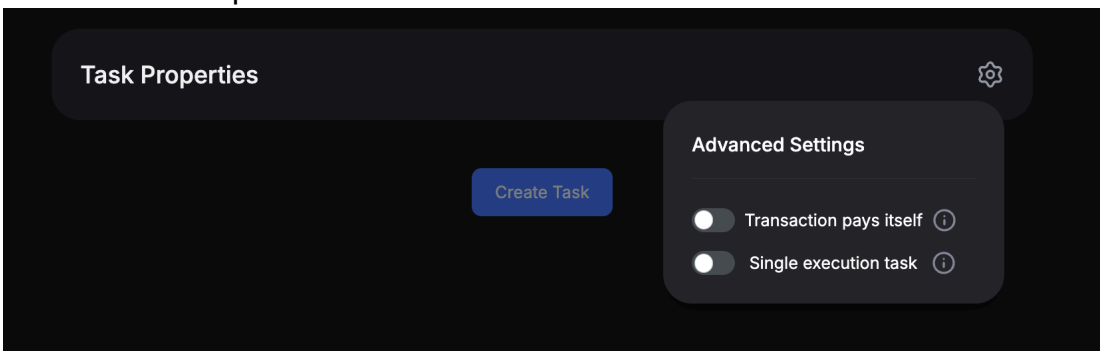
2. Network Configuration:
   Choose the blockchain network where the function should execute, such as "Berachain bArtio".

3. Task Configuration:
   If your function requires secret variables (e.g., API keys), securely enter them in the "Task Secrets" section:
   - Key: Name of the variable, e.g., `API_KEY`.
   - Value: The corresponding secret value.

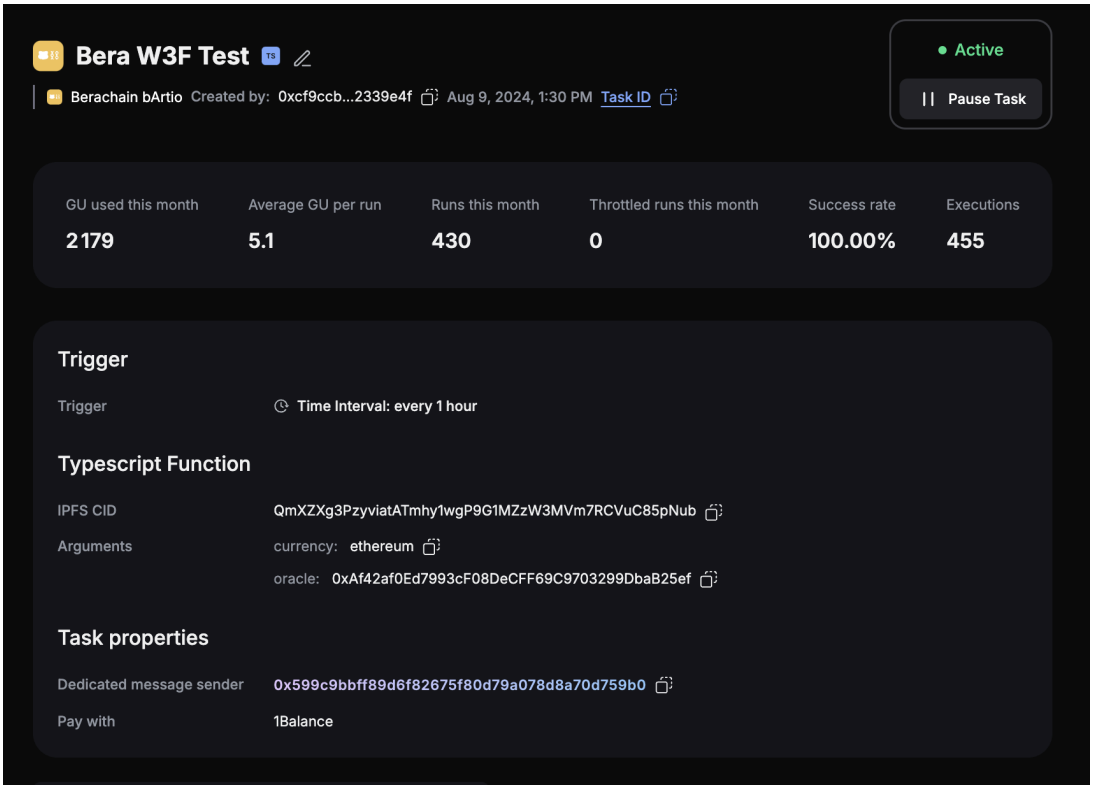
   Ensure you save each secret to guarantee its secure storage.

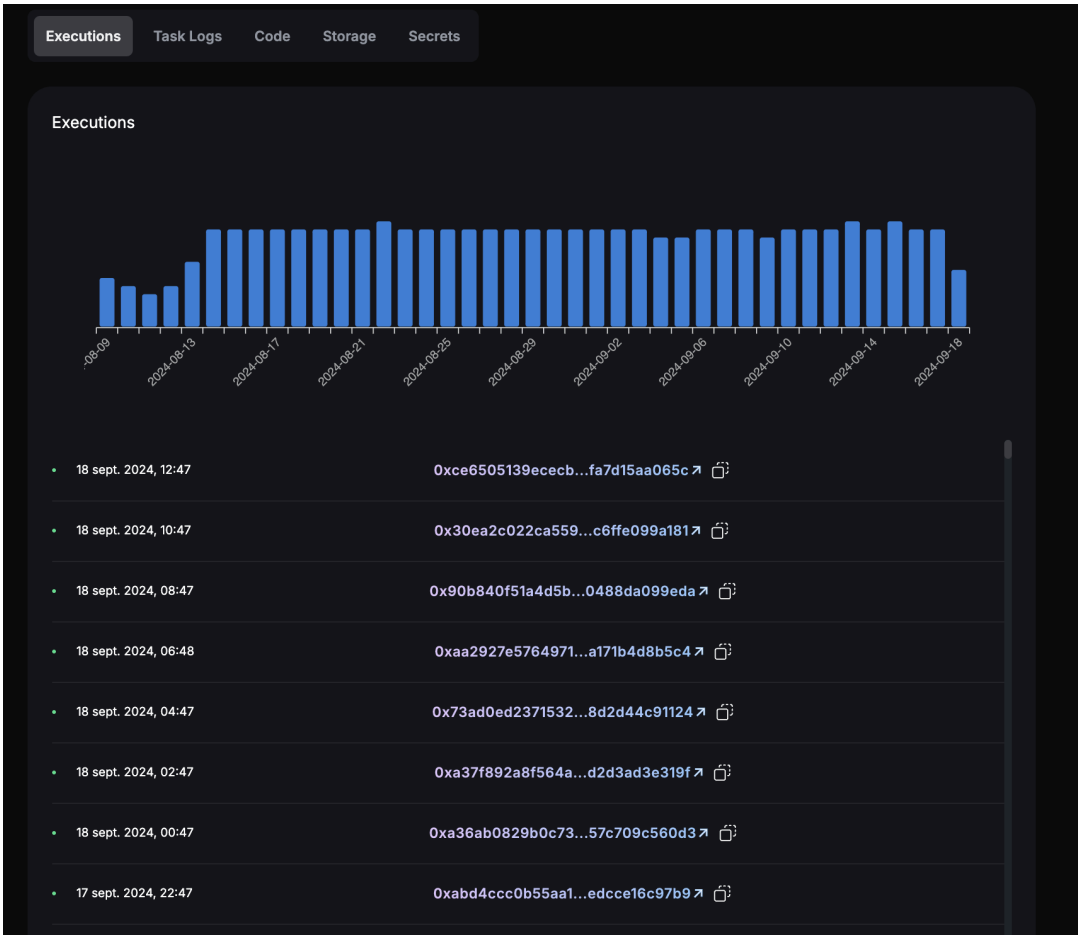
Once all the fields have been filled in, click on “Create Task”, then sign the transaction.

For more advanced settings on the UI, you can modify the task properties by clicking on the “Task Properties”



Once the task has been created, a dashboard will appear, allowing you to monitor your task and track execution and run logs





Besides the task logs available in the UI, Gelato Web3 Functions offer a more detailed and granular monitoring system providing status and logs APIs.

Provided the `ChainId` and `taskId`, this API will return the current Task status
Copy

```
None

https://api.gelato.digital/tasks/web3functions/networks/{chainId}/tasks/{taskId}/status
```

Conclusion

In summary, Gelato's Web3 Functions make it simple to automate and monitor smart contract tasks. Whether you're working with TypeScript or Solidity, the platform gives you the tools to track performance and usage in real-time. With flexible options for task creation, you can easily choose the best setup for your needs. Gelato's Web3 Functions help you streamline your workflow, so you can focus more on building and less on managing tasks.