

Rapport technique SmartContract

Boris Chan Yip Hon

Bill Zaghdoudi

Août 2024

Financement

Ce projet a été financé par le contrat de projet DistriTraca n° 2023290944 AID - Sorbonne Université.

1 Introduction

1.1 Objet du rapport

L'objectif de ce rapport est d'apporter un soutien technique aux étudiants de licence et de master qui n'ont aucune connaissance préalable des smart contracts et des technologies blockchain. L'objectif est qu'ils puissent s'appuyer sur ce document pour acquérir les compétences techniques nécessaires à la mise en œuvre de smart contracts en toute autonomie. Ce rapport est conçu pour être un guide complet et autonome, fournissant toutes les connaissances nécessaires pour déployer un contrat intelligent sur une blockchain compatible EVM (Ethereum Virtual Machine).

1.2 Context

Ce rapport a été rédigé dans le cadre d'un projet de recherche sur les technologies blockchain. L'un des objectifs du projet était la mise en œuvre de contrats intelligents sur des blockchains compatibles EVM. L'objectif de ce rapport est de documenter les étapes et les concepts impliqués dans le déploiement d'un contrat intelligent, afin de faciliter la prise en main de ce projet par les étudiants de premier et deuxième cycles qui travailleront sur ce projet à l'avenir.

1.3 Préface

1. Explication des outils et des termes techniques

2. Installation préalable

(a) MetaMask (sous-sous-section [3.1](#)) (b)

ChainList (sous-sous-section [3.2](#)) (c)

Remix (sous-sous-section [3.3](#))

(d) Code VisualStudio (sous-section 3.4)

3. Rédiger un contrat intelligent

(a) Méthodologie (sous-section 4) (b)

Remix (sous-section 4.3) i.

Solidité de la compilation (sous-section 4.3) ii.

Déploiement et exécution des transactions (sous-section 4.4)

(c) Ordre d'exécution de l'action (sous-section 4.5)

4. Automatisez les appels à l'aide de Node JS

(a) Clé privée (sous-section 5.1) (b) ABI

(sous-section 5.2) (c) Adresse

du contrat (sous-section 5.2) (d) Code Node

JS (sous-section 5.4)

5. Annexe

- Structure des données du jeu narratif
- Contrat intelligent de jeu narratif
- Code Node JS

2 Explication des outils et des termes techniques

- MetaMask : MetaMask est une extension de navigateur qui sert de portefeuille numérique pour la gestion des crypto-monnaies telles que l'Ether. Elle permet d'interagir facilement avec les applications décentralisées (dApps) sur la blockchain Ethereum. Concrètement, MetaMask peut être utilisé pour stocker, envoyer et recevoir des crypto-monnaies. Il peut également être utilisé pour signer des transactions afin de prouver que je suis bien celui qui les autorise.
- ChainList : ChainList est un site Web qui aide les utilisateurs à connecter facilement leur portefeuille MetaMask à différentes blockchains, également appelées « réseaux ». Plus précisément, lorsqu'une blockchain autre qu'Ethereum est utilisée, les informations de cette blockchain doivent être ajoutées manuellement à MetaMask. ChainList simplifie ce processus en fournissant une liste de réseaux compatibles qui peuvent être ajoutés à MetaMask en un seul clic. C'est un outil pratique pour accéder rapidement à plusieurs blockchains sans avoir à saisir manuellement des paramètres.
- Remix : Remix est un outil largement utilisé pour écrire, tester et déployer des contrats intelligents sur la blockchain Ethereum. Ce logiciel, accessible directement via un navigateur web, permet aux développeurs de coder en Solidity, le langage de programmation utilisé pour créer des contrats intelligents. En pratique, Remix est utilisé pour

Rédigez le code d'un contrat intelligent, testez-le en simulant des transactions et déployez-le enfin sur une blockchain. L'interface de Remix est conçue pour simplifier ces étapes, en proposant des outils de débogage et de simulation intégrés.

Ainsi, les développeurs peuvent s'assurer que leur code fonctionne correctement avant de le rendre opérationnel sur la blockchain.

- **Testnet** : un testnet est une version alternative d'une blockchain utilisée principalement pour tester des applications, des contrats intelligents ou des mises à jour avant leur déploiement sur la blockchain principale, appelée mainnet. Sur un testnet, des versions « factices » de cryptomonnaies sont utilisées, ce qui permet aux développeurs d'expérimenter sans risquer de perdre de vraies cryptomonnaies. Le testnet est souvent utilisé pour vérifier que tout fonctionne correctement et que les éventuels bugs ou erreurs sont corrigés, avant de déployer les modifications sur le mainnet.
- **RPC** : Un RPC (Remote Procedure Call) est une méthode utilisée pour permettre à un programme de communiquer à distance avec une blockchain. En pratique, un RPC est utilisé pour envoyer des commandes ou des requêtes d'une application, comme un portefeuille ou un logiciel de développement, à un nœud de la blockchain. Par exemple, lorsqu'un utilisateur de MetaMask souhaite consulter son solde ou envoyer une transaction, MetaMask envoie une requête RPC au nœud de la blockchain pour obtenir les informations nécessaires ou exécuter la transaction. Le RPC est donc essentiel à l'interaction entre une application et une blockchain.
- **Faucet** : un faucet est un outil qui permet aux utilisateurs de recevoir gratuitement de petites quantités de cryptomonnaie, généralement sur un réseau de test. Un faucet est principalement utilisé pour tester des applications ou des contrats intelligents sur un réseau de test. Comme les cryptomonnaies sur un réseau de test n'ont pas de valeur réelle, un faucet distribue ces fonds fictifs afin que les développeurs puissent exécuter des tests sans risquer leurs propres actifs. Les utilisateurs peuvent demander des fonds à un faucet, qui les envoie ensuite vers leur portefeuille, leur permettant ainsi d'interagir avec le réseau de test.
- **ABI** : L'ABI (Application Binary Interface) est un élément essentiel pour interagir avec un contrat intelligent sur une blockchain. L'ABI décrit les fonctions et les structures de données d'un contrat intelligent de manière standardisée.
Lorsqu'une application ou un utilisateur souhaite interagir avec un contrat intelligent, l'ABI est utilisé pour comprendre comment appeler les fonctions du contrat, quels paramètres sont requis et comment interpréter les données renvoyées. En bref, l'ABI sert de « manuel d'instructions » qui permet à d'autres applications de communiquer correctement avec un contrat intelligent.

Voici comment les différents concepts sont interconnectés :

- **MetaMask est interconnecté avec** :
 - **ChainList** : pour ajouter facilement de nouveaux réseaux blockchain à MetaMask
 - **RPC** : pour permettre à MetaMask de communiquer avec différentes blockchains nœuds.

- Testnet : MetaMask peut être configuré pour interagir avec les réseaux de test afin de tester les contrats intelligents.
- Faucet : utilisé pour obtenir des crypto-monnaies sur un testnet via MetaMask.
- Remix est interconnecté avec :
 - Testnet : pour tester les contrats intelligents développés dans Remix avant de les déployant sur le réseau principal.
 - MetaMask : pour signer et déployer des contrats intelligents directement depuis Remix.
- Testnet est interconnecté avec :
 - Faucet : pour obtenir des crypto-monnaies de test à utiliser sur le testnet.
- RPC est interconnecté avec :
 - Remix : pour déployer des contrats intelligents via des appels RPC à la blockchain.
- ABI est interconnecté avec :
 - Remix : pour générer l'ABI d'un contrat intelligent qui sera utilisé pour interagir avec ce contrat.
 - MetaMask : pour comprendre et interagir avec les fonctions d'un contrat intelligent déployé.

Le lien entre MetaMask, RPC, Testnet, Faucet, Remix et ChainList peut s'expliquer comme suit :

- ChainList et MetaMask : MetaMask est un portefeuille numérique qui permet aux utilisateurs de gérer leurs crypto-monnaies et d'interagir avec des applications décentralisées sur différentes blockchains. Cependant, pour interagir avec une blockchain spécifique via MetaMask, il faut d'abord ajouter les paramètres de cette blockchain (comme l'adresse du nœud RPC, l'ID réseau, etc.). ChainList, en revanche, facilite cette tâche. Il fournit une liste de blockchains compatibles que les utilisateurs peuvent ajouter directement à MetaMask en un seul clic.

De cette façon, ChainList simplifie le processus de configuration de nouveaux réseaux sur MetaMask, permettant à l'utilisateur d'accéder rapidement à différentes blockchains sans avoir à saisir manuellement les informations techniques nécessaires.

- MetaMask et RPC : MetaMask utilise les RPC (Remote Procedure Calls) pour communiquer avec les différents nœuds de la blockchain. Lorsqu'un utilisateur souhaite envoyer une transaction, consulter un solde ou interagir avec un smart contract via MetaMask, une requête RPC est envoyée au nœud de la blockchain pour exécuter cette action. Le RPC permet ainsi à MetaMask de se connecter à la blockchain, d'envoyer des instructions et de recevoir des données en retour.
- MetaMask et Testnet : MetaMask peut être configuré pour se connecter à un testnet, qui est une version de la blockchain utilisée pour les tests. En utilisant

MetaMask sur un réseau de test, les développeurs et les utilisateurs peuvent expérimenter des transactions ou des contrats intelligents sans utiliser de véritables crypto-monnaies. Cela leur permet de vérifier que tout fonctionne correctement avant de déployer sur la blockchain principale (mainnet).

- **MetaMask et Faucet** : Lorsqu'un utilisateur souhaite tester sur un testnet via Meta-Mask, il a besoin de cryptomonnaies spécifiques à ce testnet, qui n'ont pas de valeur réelle. Un faucet permet d'obtenir gratuitement ces cryptomonnaies de test.
L'utilisateur peut demander des fonds via un faucet, qui envoie ces crypto-monnaies directement vers son portefeuille MetaMask, lui permettant de tester des transactions ou des contrats intelligents sur le testnet.
- **RPC et Remix** : Remix, en tant qu'environnement de développement pour les contrats intelligents, utilise également RPC pour déployer et tester des contrats sur une blockchain. Lorsqu'un développeur déploie un contrat intelligent à partir de Remix, une requête RPC est envoyée à un nœud de la blockchain pour enregistrer le contrat sur le réseau. De plus, des actions telles que la lecture ou l'exécution des fonctions d'un contrat intelligent sont également effectuées via des appels RPC.
- **Interaction entre MetaMask et Remix via RPC** : Lorsqu'un développeur utilise Remix pour déployer un contrat intelligent, il peut choisir de signer la transaction via MetaMask. Dans ce cas, Remix génère un appel RPC pour déployer le contrat, et MetaMask intervient pour signer et autoriser cette requête avant qu'elle ne soit envoyée au nœud de la blockchain. MetaMask garantit ainsi que les actions initiées dans Remix sont sécurisées et validées par l'utilisateur via RPC.

3 Installation prérequis

3.1 MétaMasque

Extension de plugins MetaMask, à installer directement sur le navigateur. Installer MetaMask sur le navigateur, via une extension (figure 1) : <https://chromewebstore.google.com/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn?hl=fr>



Figure 1 : extension Chrome

Remarque : au lieu de « Supprimer de Chrome », il devrait être indiqué « Ajouter à Chrome » (ou quelque chose de similaire). (J'ai déjà installé MetaMask, d'où la suppression

message) ou utilisez le lien suivant, selon votre support (figure 2). <https://metamask.io/download/>

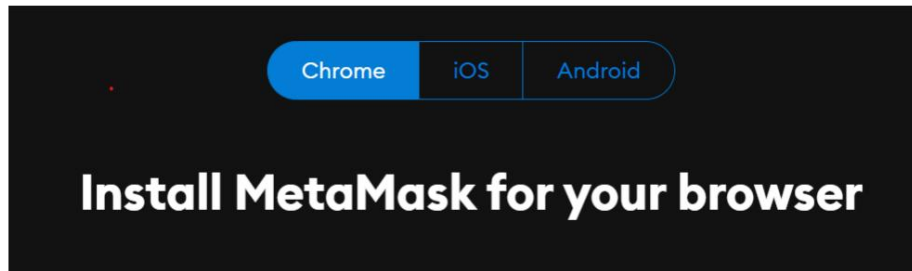


Figure 2 : Différents supports d'installation

Une fois MetaMask installé sur votre navigateur, il apparaît par défaut dans le coin supérieur droit. MetaMask est représenté par une tête de renard (figure 3).

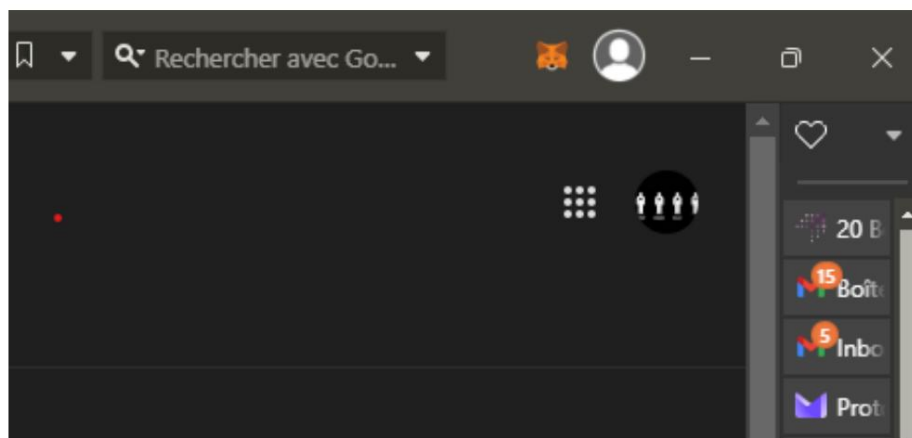


Figure 3 : Icône MetaMask

Voici la fenêtre, une fois déployée. Suivez les instructions pour créer un compte. Documentation officielle <https://support.metamask.io/getting-started/getting-started-with-metamask/> ou suivez les étapes pour créer un compte directement depuis la fenêtre ci-dessus.(figure 4).

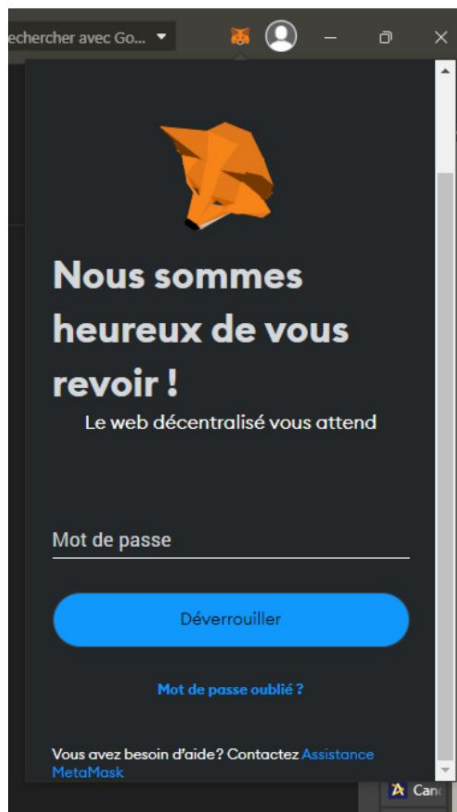


Figure 4 : Fenêtre MetaMask

La fenêtre d'ouverture de MetaMask affiche les informations suivantes :

- nom actuel du mainnet ou du testnet
- portefeuille sélectionné
- adresse actuelle du portefeuille
- le montant de la crypto-monnaie sur le compte
- quelques fonctionnalités :
 - acheter
 - envoyer
 - échanger
 - pont -
 - historique du portefeuille

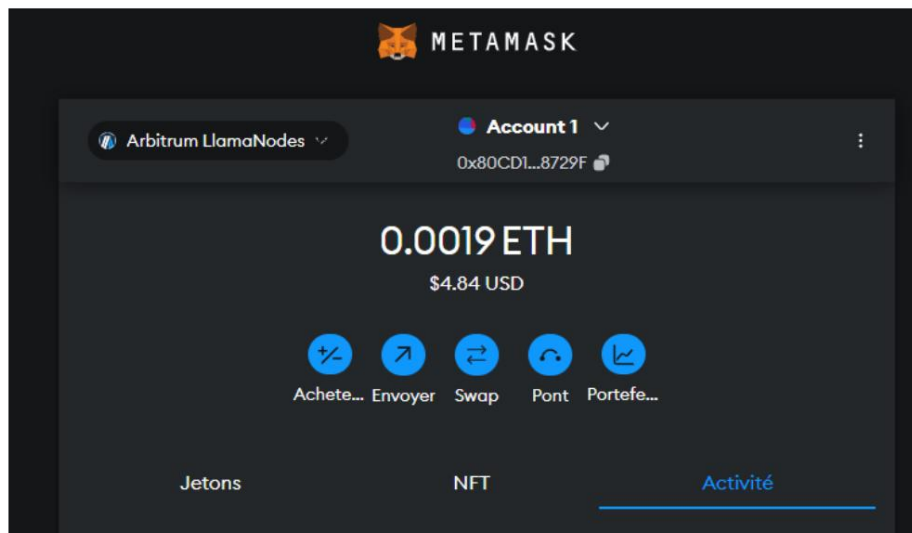


Figure 5 : Configuration de la page MetaMask

3.2 Liste de chaînes

Ajoutez des blockchains et leur réseau de test respectif, si possible, intégrez-y des cryptomonnaies ou testez des échantillons. Lien vers ChainList : <https://chainlist.org/>

Une fois sur la page ChainList, une barre de recherche vous permet de cibler les blockchains que vous souhaitez ajouter à votre compte MetaMask. (figure 6).

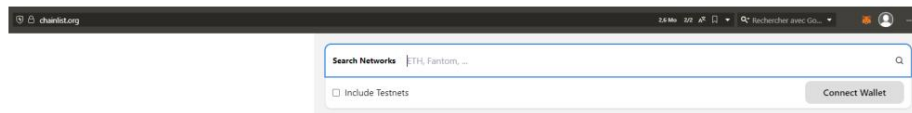


Figure 6 : Barre de recherche ChainList

Cliquez sur « Connecter le portefeuille » pour connecter votre compte MetaMask. La fenêtre de confirmation Meta-Mask apparaîtra, vous permettant de vous connecter à votre compte Meta-Mask.

Dans la capture d'écran suivante, je souhaiterais ajouter une chaîne de la blockchain Arbitrum à mon compte MetaMask, si possible, le mainnet de la blockchain et le testnet associé. (figure 7). Dans le cas présent, où plusieurs chaînes sont présentées, il faut d'abord rechercher la blockchain que l'on souhaite ajouter.

Ceci étant dit, il existe plusieurs chaînes car elles peuvent être une version antérieure

avant une fusion, soit les chaînes utilisent des algorithmes de consensus différents, soit un changement de version ou pour une autre raison. Dans ce cas, il est nécessaire de faire quelques recherches pour identifier celui dont vous avez besoin. Les mainnets nécessitent la crypto-monnaie de la blockchain pour pouvoir effectuer des transactions sur eux.

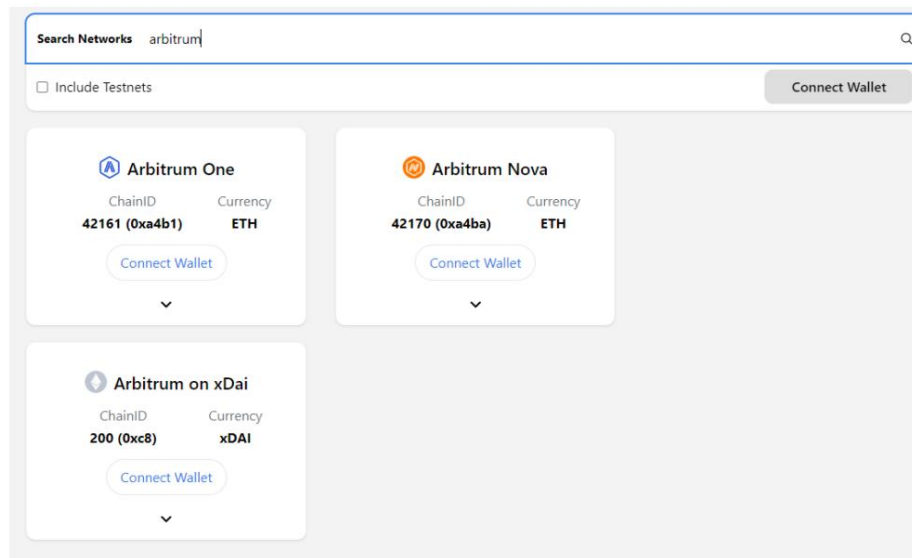


Figure 7 : Recherche d'Arbitrum sans réseaux de test

3.2.1 Réseau de test

Il existe une alternative gratuite : utiliser des chaînes de test de blockchain publiques, Testnets. Dans la capture d'écran suivante, j'ai coché la case « Inclure les Testnets ». Les Testnets utilisent des faucets pour les tests, pas des crypto-monnaies. De même, il existe plusieurs Testnets disponibles, vous devrez donc faire quelques recherches pour savoir à quel mainnet un testnet est rattaché. (figure 8).

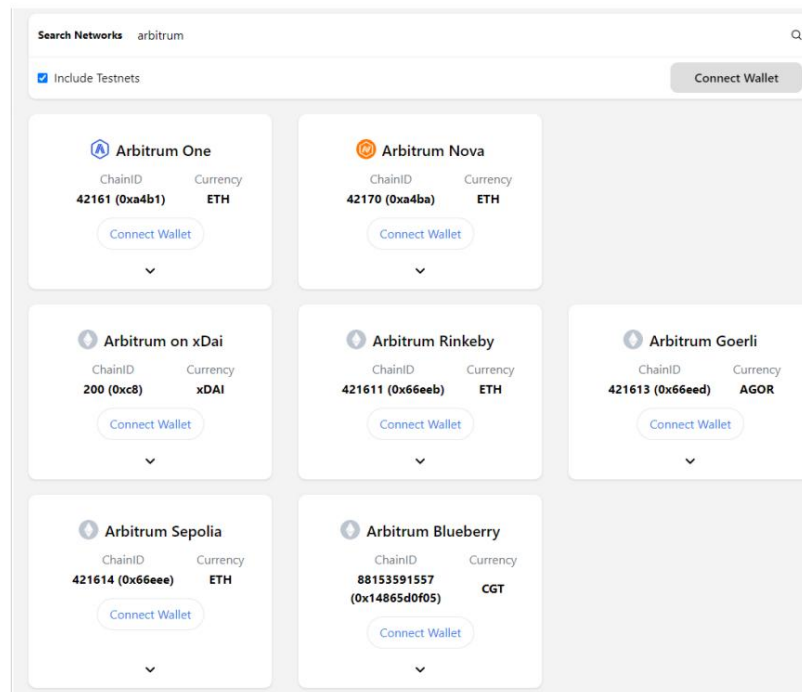


Figure 8 : Recherche d'Arbitrum avec des réseaux de test

Dans les deux cas, mainnets ou testnets, cliquez sur « Connect Wallet » pour ajouter le canal à votre compte MetaMask. Si vous ne l'avez pas déjà fait, vous devrez demander une connexion à MetaMask. Une fenêtre apparaît, demandant la confirmation du changement de réseau. Une fenêtre similaire à la capture d'écran suivante (figure 9). Autoriser, pour finaliser le changement de chaîne.

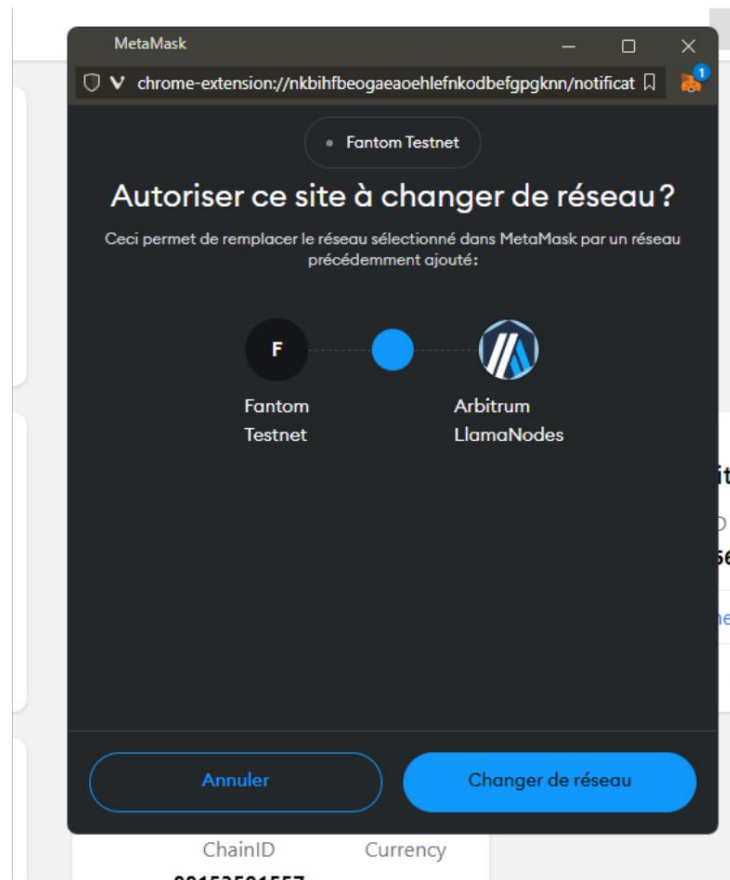



Figure 9 : Demande de changement de chaîne

Il est également judicieux de conserver le RPC que vous utilisez et de le lier au fichier JS lors de l'automatisation des appels. Le choix du RPC dépend de ses performances en termes de latence et de confidentialité. Pour trouver le RPC, cliquez sur la flèche pointant vers le bas dans le carré de la chaîne souhaitée - par exemple, sur la chaîne Arbitrum One.

Search Networks arbitrum

☒ Include Testnets


0x80cd...729f

 Arbitrum One

ChainID
42161 (0xa4b1)

Currency
ETH

Add to Metamask

 Arbitrum Nova

ChainID
42170 (0xa4ba)

Currency
ETH

Add to Metamask

Arbitrum One RPC URL List

II Sorting

RPC Server Address	Height	Latency	Score	Privacy	
https://arbitrum.llamarpc.com	244923592	0.153s	✓	✓	Add to Metamask
https://arbitrum-one.public.blastapi.io	244923634	0.102s	✓	⚠	Add to Metamask
https://arbitrum-one-rpc.publicnode.com	244923634	0.128s	✓	✓	Add to Metamask
https://arbitrum-one-publicnode.com	244923630	0.128s	✓	○	Add to Metamask
wss://arbitrum-one-publicnode.com	244923596	0.267s	✓	○	
https://arbitrum.drpc.org	244923592	0.298s	✓	✓	Add to Metamask
https://arb-polk.nodes.app	244923592	0.311s	✓	✓	Add to Metamask
https://arbitrum.blockpi.network/v1/rpc/public	244923592	0.343s	✓	⚠	Add to Metamask
https://arb-mainnet.g.alchemy.com/v2/demo	244923592	0.375s	✓	✓	Add to Metamask
https://api.stateless.solutions/arbitrum-one/v1/demo	244923591	0.195s	✓	✓	Add to Metamask
https://arbitrum.meowrpc.com	244923590	0.327s	✓	✓	Add to Metamask
https://arb1.arbitrum.io/rpc	244923590	0.449s	✓	○	Add to Metamask
https://endpoints.omniatech.io/v1/arbitrum/one/public	244923590	0.473s	✓	✓	Add to Metamask
https://arbitrum.rpc.subquery.network/public	244923589	0.310s	✓	○	Add to Metamask
https://rpc.ankr.com/arbitrum	244923589	0.339s	✓	⚠	Add to Metamask

Figure 10 : Liste de chaînes RPC d'Arbitrum

Il en va de même pour toutes les autres chaînes accessibles sur ChainList. Parfois, tous les liens RPC ne sont pas accessibles. Dans ce cas, utilisez les moteurs de recherche Google ou Infura. <https://www.infura.io/networks>

Attention : vous pouvez recevoir un message d'erreur indiquant que le nœud ne peut pas être atteint lors de l'exécution du fichier de nœud sur VisualStudioCode. Dans ce cas, la modification de l'adresse RPC peut résoudre le problème. J'expliquerai où modifier le RPC plus tard, dans le code.

Pour changer la chaîne, assurez-vous d'abord que la chaîne a été ajoutée à MetaMask. Si ce n'est pas le cas, revenez à l'étape ChainList. Sélectionnez ensuite simplement une autre chaîne dans le menu MetaMask. Pour déplier toutes les chaînes ajoutées à MetaMask, cliquez sur la flèche à côté du nom de la chaîne et pointant vers le bas pour déplier le menu.

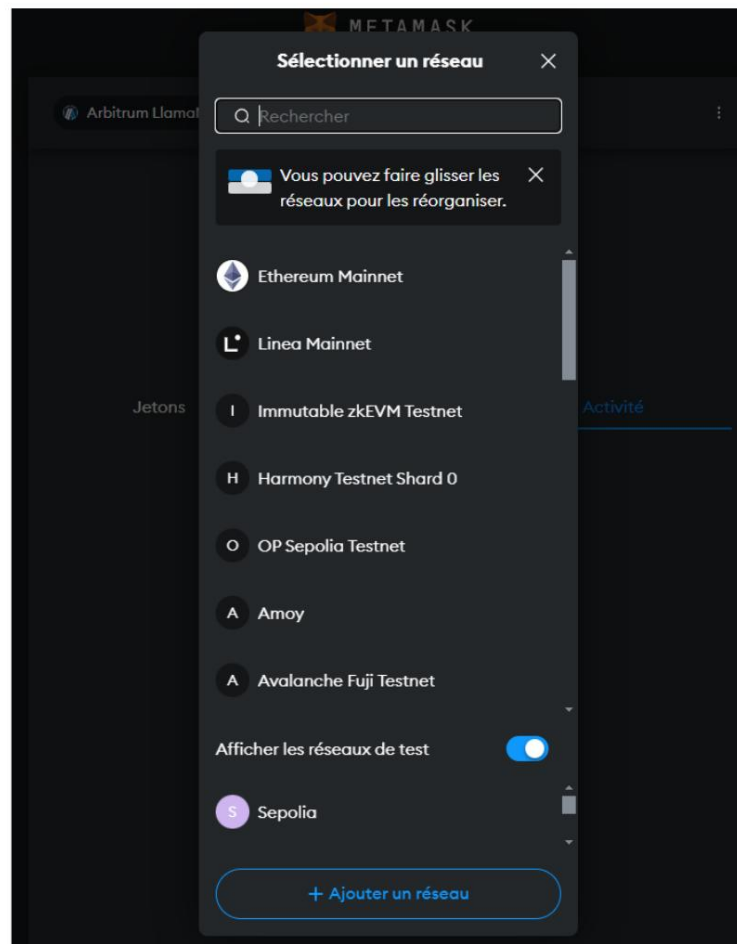


Figure 11: Chaîne change

Sélectionnez la chaîne à utiliser pour la substitution, et confirmez la fenêtre de confirmation de MetaMask, une fenêtre similaire à celle de configuration d'une chaîne sur MetaMask.

3.2.2 Robinet

Connaissez l'adresse de votre portefeuille car elle est nécessaire pour faire des demandes de faucet.

Recherche Google : [jblockchain](#)

name, faucet Quelques liens utiles, regroupant plusieurs chaînes :

<https://www.alchemy.com/faucets/ethereum-sepolia> [https://](https://faucet.quicknode.com/drip)

[faucet.quicknode.com/drip](https://www.infura.io/faucet/sepolia) [https://www.infura.io/](https://www.infura.io/faucet/sepolia)

[faucet/sepolia](https://www.infura.io/faucet/sepolia)

<https://faucets.chain.link/sepolia>

3.3 Remix

Remix propose un environnement interactif dans lequel vous pouvez écrire et tester du code en temps réel.

Ouvrez votre navigateur Web et accédez au lien suivant : <https://remix.ethereum.org/>

Remix est directement utilisable, sans

avoir besoin d'installer de bibliothèques. Le langage de programmation est Solidity, dont la syntaxe est similaire à celle de JavaScript. Vous pouvez créer plusieurs espaces de travail dédiés pour différents projets, voire tous les smart-contracts de l'espace de jeu. Les smart-contracts se trouvent dans le dossier contracts, 'narrative_game_state.sol' est le nom du contrat que j'ai utilisé pour ma structure de données. (figure 12). –

Important : les smart-contracts écrits directement dans Remix via le navigateur web sont stockés dans le cache du navigateur. Nous vous recommandons vivement de les sauvegarder systématiquement dans un fichier texte séparé.

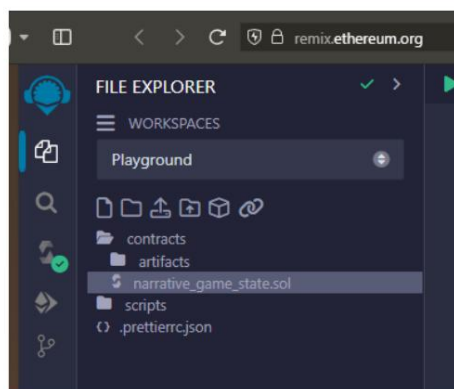


Figure 12 : Explorateur de fichiers Remix

Pour apprendre Solidity de manière plus technique et ludique, je vous conseille de suivre les tutoriels CryptoZombies. <https://cryptozombies.io/fr>

3.3.1 Compilation

Pour être averti d'éventuelles erreurs lors de la compilation, cochez la case « Compilation automatique ». Pour le choix du compilateur, le plus récent sera choisi par défaut. Il peut arriver que certaines chaînes ne fonctionnent pas avec une version de compilateur trop récente, auquel cas il faudra sélectionner une version supportable plus ancienne ou changer de version EVM de Remix

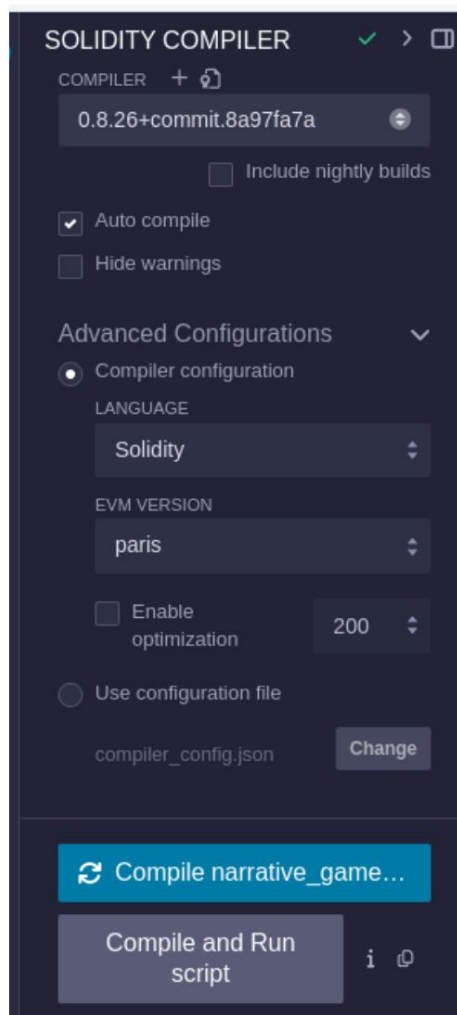


Figure 13 : Compilation automatique du remix

3.4 Code Visual Studio

VisualStudioCode est efficace et simple de prise en main. <https://code.visualstudio.com/>
L'utilisation d'un IDE simplifie la programmation des API et l'installation des bibliothèques Node.js et NPM. Pour installer Node.js et npm sur votre système, voici les commandes spécifiques à votre système d'exploitation :

- Pour Windows

– Accédez au site officiel de Node.js : <https://nodejs.org/> –

Téléchargez le programme d'installation LTS (Long Term Support) recommandé pour Fenêtres.

- Exécutez le programme d'installation téléchargé.
- Suivez les instructions à l'écran. Cela installera les deux Node.js et npm automatiquement.

- Pour Linux

- Ouvrez un terminal et exécutez les commandes suivantes :

```
1 $ curl -fsSL https://deb.nodesource.com/setup_10.x |
   sudo -E bash -
2 $ sudo apt - install nodejs
```

Après l'installation, vous pouvez vérifier que tout fonctionne correctement en exécutant les commandes suivantes dans votre terminal :

```
1 $ node -v
2 npm -v
```

4 Rédiger un contrat intelligent

4.1 Analyse de la structure des données

La première étape consiste à analyser les données sur papier. Cette analyse permet d'identifier les éléments nécessaires pour former l'essence de la structure de données du contrat intelligent.

Cela implique de comprendre les exigences du projet et de définir clairement les variables, les types de données et les fonctions qui seront implémentés dans le contrat intelligent.

4.2 Programmation

Après avoir décrit la structure de données avec les variables et les structures sur papier (ou dans un fichier texte), le smart-contract peut être codé dans Solidity. Pour coder le smart-contract, le moyen le plus simple est de lancer une requête vers ChatGPT pour obtenir le code Solidity associé à la structure de données que nous avons identifiée précédemment :

Écrivez le contrat intelligent associé à la structure de données suivante : « copie » et collez la structure de données »

Prenons comme exemple un contrat intelligent qui prend un nom en entrée et renvoie « Bonjour » + nom + « , c'est un plaisir de vous rencontrer ! ». La structure des données est la suivante.

```
1 struct Salutation { chaîne nom ;
2
3 }
```

Le contrat intelligent associé, généré par ChatGPT.

```
1 // SPDX - Licence - Identifiant : MIT 2 pragma solidity ^0.8.0;
3
```



```

4 contrat Salutation {
5
6     structure Nom {
7         nom de chaîne ;
8     }
9
10    fonction greet ( chaîne mémoire _name ) public pure renvoie ( chaîne
        mémoire ) {
11        Mémoire du nom personne = Nom ({
12            nom : _nom
13        });
14
15        chaîne mémoire salutation = chaîne ( abi . encodePacked (" Bonjour ",
            personne . nom , ", c'est un plaisir de vous rencontrer"));
16
17        retour salutation ;
18    }
19 }

```

Liste 1 : Smart contract en Solidity

Ce code doit être structuré de manière à implémenter correctement les fonctionnalités définies lors de l'analyse. Si ce n'est pas le cas, modifiez les code utilisant ChatGPT.

Pour apprendre à traduire des structures de données en contrats intelligents, il est important de se familiariser avec plusieurs concepts de base de la programmation Solidity, comme ainsi que d'acquérir une solide compréhension des contrats intelligents et de la blockchain en général. Voici quelques ressources pédagogiques qui peuvent vous aider à maîtriser cette compétence :

<https://cryptozombies.io/>

<https://solidity-by-example.org/>

<https://docs.soliditylang.org/>

4.3 Compilation

« Injector Provider » est directement lié à la blockchain actuelle utilisée sur Meta-Mask. C'est ici que vous spécifiez quelle chaîne sera utilisée par Remix pour déployer le contrat intelligent. Si la chaîne change entre-temps, l'environnement doit à mettre à jour.

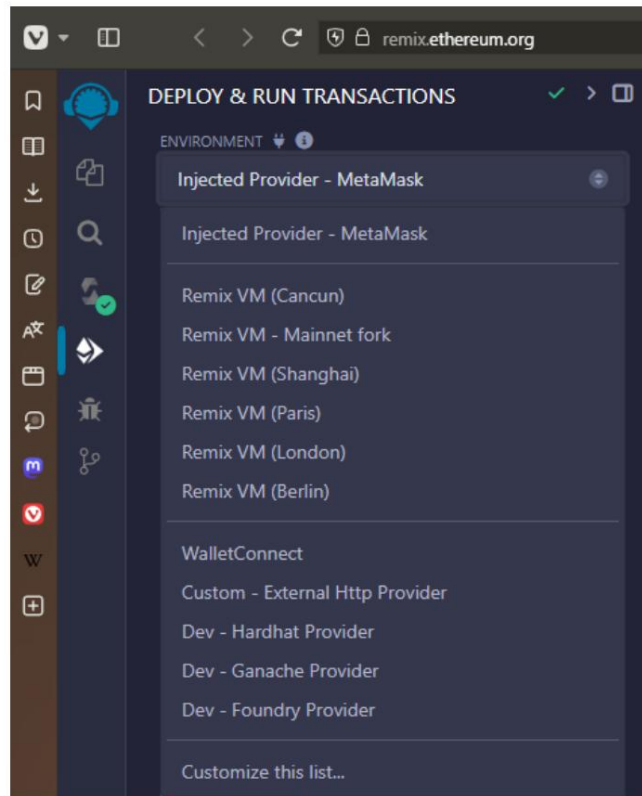


Figure 14 : Remixage de l'injecteur

4.4 Déploiement

Voici une capture d'écran de la barre des tâches de déploiement de Remix. Vous devez vérifier cela :

- le contrat intelligent est compilé correctement
- le fournisseur d'injecteurs a sélectionné MetaMask
- les informations spécifiées dans le compte sont correctes : adresse du portefeuille sur Meta-Mask et le montant de crypto-monnaie ou de faucets sur le compte.

Si les informations ne sont pas correctes, mettez à nouveau à jour les informations d'environnement. Une fois coché, cliquez sur « Déployer ». (figure 15).

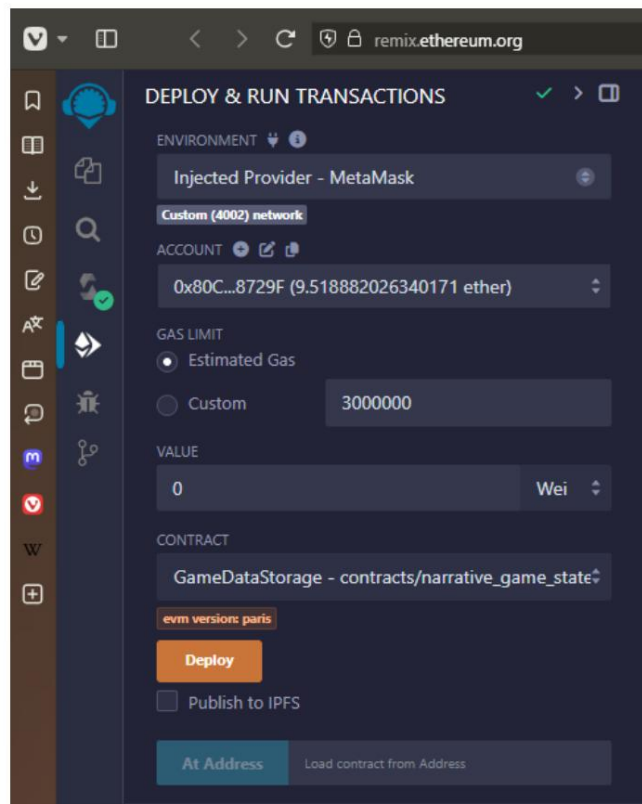


Figure 15 : Déployer Remix

Une confirmation du déploiement du smart-contract sur la chaîne et à quel coût doit être validé pour approuver le déploiement. (figure 16).

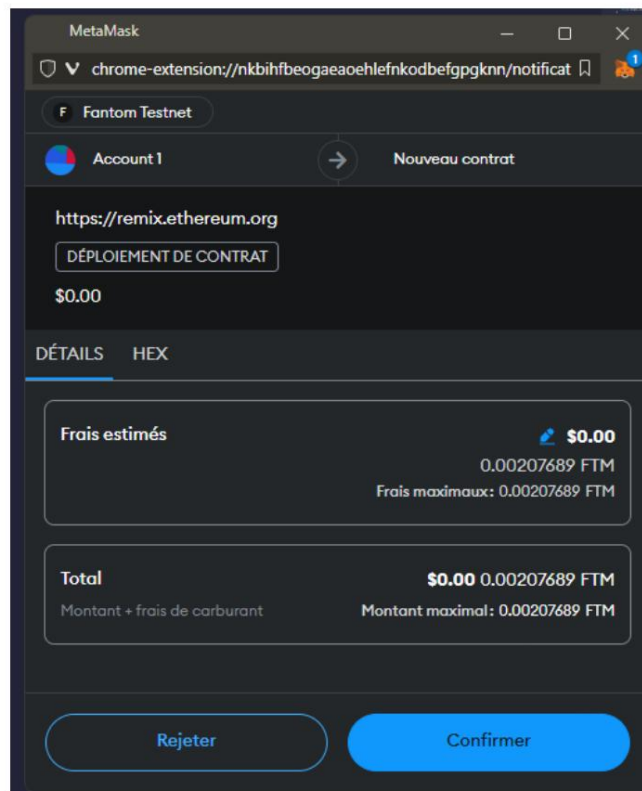


Figure 16 : Remix de la demande de déploiement

Dans l'exemple de salutation, la figure 17 décrit la réponse au message intelligent.
Contrat Demande de déploiement de salutation.

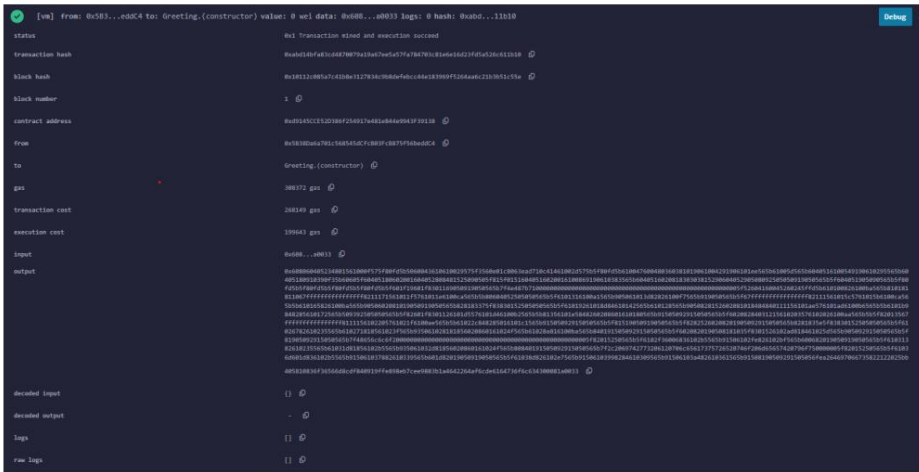


Figure 17 : Déploiement du message d'accueil

Une fois le contrat déployé, il est possible d'interagir directement avec lui via Remix (figure 18). Les paramètres sont saisis manuellement, ce qui permet de vérifier que le contrat s'exécute correctement avant de l'exécuter sur une API.

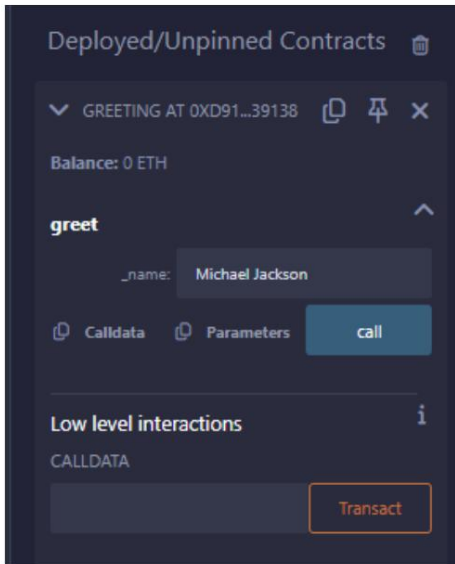


Figure 18 : Transmission manuelle des paramètres

Dans cet exemple, je saisis le nom « Michael Jackson » comme paramètre, puis j'appelle le smart-contract en cliquant sur « call ». Enfin, la sortie du smart-contract

contrat (figure 19) :



Figure 19 : Salutations à Michael Jackson

4.5 Ordre d'exécution des actions

Pour résumer les étapes :

1. écrire toute la structure de données à évaluer (variable, structure, énumération, etc.) sur une feuille de papier
2. codez vous-même le contrat intelligent en utilisant les exemples ou utilisez ChatGPT coder le contrat intelligent associé à la structure de données
3. ouvrir une fenêtre de remix
4. copier/coller le code dans le projet Remix créé à cet effet
5. Activer la compilation automatique de Remix
6. choisissez le compilateur pris en charge (par défaut, le compilateur sera le plus récent)
7. Connectez-vous à votre compte MetaMask
8. ajoutez la chaîne à tester à MetaMask via ChainList
9. changer de chaîne si nécessaire, pour avoir la bonne chaîne à tester
10. mettre en place le bon environnement, c'est à dire l'injecteur MetaMask, ou choisir l'un des environnements proposés localement pour exécuter les tests
11. Déploiement de smart-contract
12. interagissez avec le contrat intelligent en saisissant des valeurs de paramètres pour garantir que le contrat intelligent s'exécute comme requis.

5 Automatiser les appels à l'aide de Node JS

5.1 Clé privée

Les détails du compte sont disponibles dans « Détails du compte », après avoir fait défiler les 3 points bar. L'adresse du portefeuille et la clé privée sont incluses.

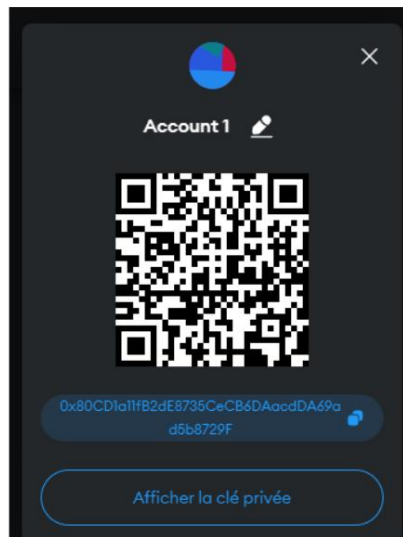


Figure 20 : Clé privée

Une fois le smart-contract déployé, Remix fournit l'ABI (Application Binary Interface) et l'adresse du contrat déployé. Ces éléments sont essentiels pour interagir avec le smart-contract à partir d'applications externes. L'ABI définit l'interface de communication avec le smart-contract,

tandis que l'adresse est utilisée pour localiser le contrat intelligent sur la blockchain.

5.2 ABI

Le code ABI est directement disponible en cliquant sur « ABI », une copie de l'ABI est réalisée et prêt à être collé comme indiqué sur la figure 21.

Pour coller, l'ABI remplace ["contenu"] par la copie de l'ABI. Par défaut, la copie comprend les crochets ouvrants et fermants. Important : une erreur courante consiste à ne pas sélectionner les crochets lors du collage de l'ABI.

résultat, l'ABI ressemble à ceci : [["contenu"]] ce qui peut conduire à une compilation erreurs.

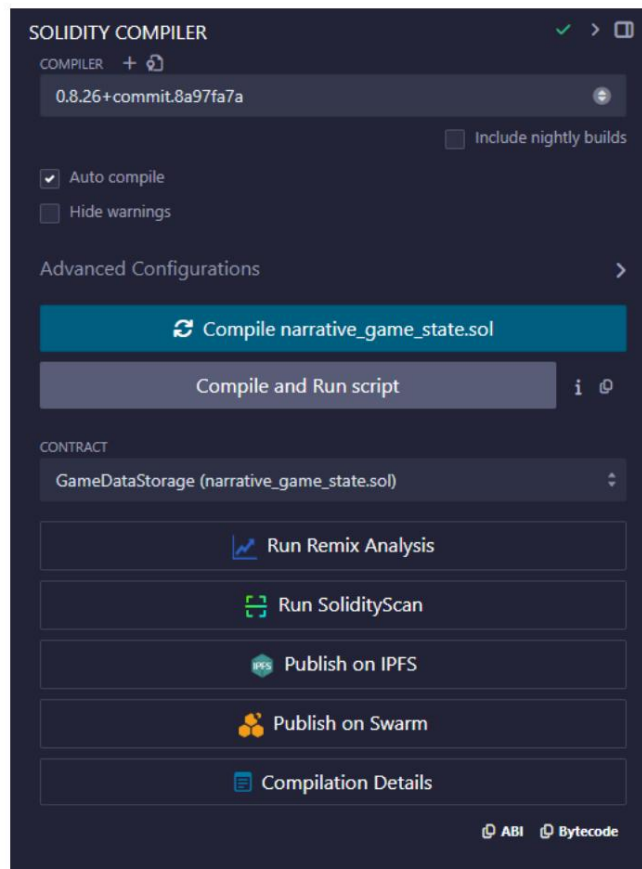


Figure 21 : Remix ABI

5.3 Adresse du contract

Les contrats intelligents déployés sont référencés dans « Contrats déployés/désépinglés », où vous pouvez copier/coller les adresses.

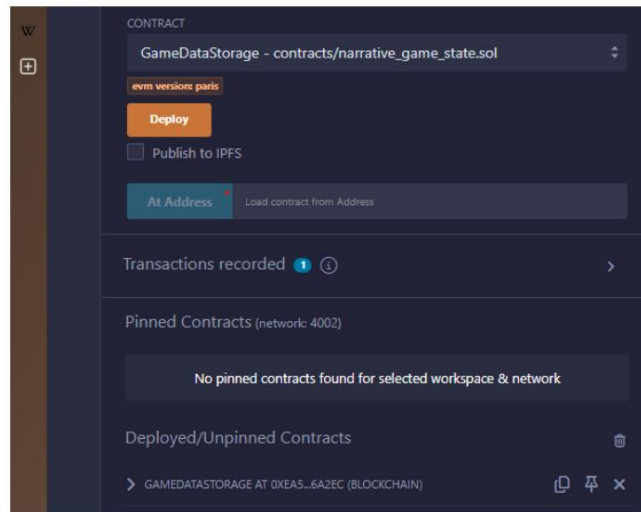


Figure 22 : Contrats déployés non épinglés

5.4 Code Node JS

Le code JS associé à l'automatisation des requêtes est ci-dessous. Les éléments requis sont la clé privée du portefeuille du compte MetaMask, qui sera utilisé pour les smart-contracts, l'adresse RPC et l'adresse du smart-contract une fois déployé.

Le code en question est un script JavaScript qui utilise la bibliothèque Web3 pour interagir avec un contrat intelligent déployé sur une blockchain compatible Ethereum. Le script effectue plusieurs actions, notamment la signature de messages, l'envoi de transactions à un contrat intelligent et la vérification des signatures. Voici une explication détaillée du code :

```
1 const { Web3 } = require (" web3 "); 2 const fs = require ("
fs "); 3 const util = require (" util ");
```

Listing 2: Code JS utilis'e pour les r'equetes

Importer des modules

- 'web3' : bibliothèque utilisée pour interagir avec la blockchain Ethereum.
- 'fs' : module de système de fichiers pour la lecture et l'écriture de fichiers.
- 'util' : module utilitaire pour le formatage des sorties.

```
1 // Initialiser Web3 2 const web3 =
new Web3 ( " mettre le lien RPC ici ");
```

Initialisation de Web3

- Le script initialise un objet Web3 en se connectant à un fournisseur RPC (Remote Procedure Call) sur la blockchain. Vous devez remplacer « mettre le lien RPC ici » par l'URL du fournisseur RPC (tel que Infura, Alchemy, etc.).

```
1 // La clé privée de votre compte (assurez-vous de la garder en sécurité) 2 const privateKey = mettre
sa clé privée ici"; 3 const account = web3 . eth . accounts . privateKeyToAccount
( privateKey ); 4 web3 . eth . accounts . wallet . add ( account ); 5 web3 . eth . defaultAccount = account . address ;
```

Gestion des comptes

- Le compte de l'utilisateur est récupéré à partir d'une clé privée qui est convertie en compte Web3. La clé privée doit être sécurisée et ne jamais être exposée au public.
- Le compte est ensuite ajouté au portefeuille Web3 afin qu'il puisse signer des transactions. actions et est défini comme compte par défaut.

```
1
2 // ABI de votre smart contract 3 const contractABI = [ "
copier / coller l'ABI ici [] . "];
, soyez prudent avec le
4
5 // L'adresse de votre contrat déployé 6 const contractAddress = "mettez
l'adresse du contrat ici ";
```

Contrat intelligent

- L'ABI (Application Binary Interface) du contrat intelligent est un tableau JSON qui décrit l'interface du contrat (ses fonctions, ses événements, etc.). Il doit être copié au lieu de « copier/coller l'ABI ici, faites attention aux [] ».
- L'adresse du contrat déployé doit être précisée au lieu de « mettre l'adresse du contrat ici ».

```
1 const contract = nouveau web3 . eth . Contract ( contractABI
, contratAdresse
);
```

Créer une instance de contrat intelligent Ethereum :

- « web3.eth.Contract » fait partie de « web3.js », qui est une bibliothèque utilisée pour interagir avec la blockchain Ethereum.
- « contractABI » : L'ABI (Application Binary Interface) définit les méthodes et les événements du contrat intelligent Ethereum.
- « contractAddress » : l'adresse du contrat intelligent déployé sur le chaîne de blocs.

- Cette ligne crée une instance de contrat, qui vous permet d'interagir avec les fonctions, les événements et les données du contrat intelligent.

```
1 const logFile = fs . createWriteStream (" nom_fichier .log", { flags : "a"
  });
2 const logStdout = processus . stdout ;
```

Journalisation dans le fichier et la console :

- « fs.createWriteStream » : ouvre un fichier appelé « nom du fichier.log », pour écrire les journaux à. Les journaux « flags: » a signifie que le fichier est ouvert en mode ajout (donc nouveau « » sont ajoutés à la fin du fichier, plutôt que de l'écraser).
- « process.stdout » : il s'agit de la sortie par défaut (généralement le terminal) pour « console.log »

```
1 console . log = fonction () {
2   fichierjournal.write(util.format.apply(null, arguments) + "\n");
3   logStdout.write(util.format.apply(null, arguments) + "\n");
4 }
```

Cela redéfinit le comportement de « console.log »

- « util.format.apply(null, arguments) » : prend les arguments passés à « console.log » et les formate sous forme de chaîne.
- Cette chaîne formatée est écrite dans le fichier journal (« logFile.write(...) ») et le terminal ("logStdout.write(...)",) donc les journaux sont enregistrés dans les deux lieux

```
1 console . erreur = fonction () {
2   fichierjournal.write(util.format.apply(null, arguments) + "\n");
3   logStdout.write(util.format.apply(null, arguments) + "\n");
4 }
```

De même, « console.error » est redéfini de la même manière que « console.log », garantissant que les erreurs sont également enregistrées dans le fichier et dans le terminal.

```
1 fonction asynchrone createEvents() {
2   pour ( soit i = 0; i < 100; i ++ ) {
```

La fonction « createEvents » génère et envoie des transactions au contrat intelligent dans une boucle (100 fois dans cet exemple).

```
1
2   const UUID = const_ids ' exemple - uuid - playerId -${i} ' ;
3   = {
4     ID_joueur : ' ID joueur -${i} ',
5     ID_jeu : ' OMFG ',
6     ID_joueur_signé : ' no_id_player ',
7     ID_jeu_signé : ' OMFG '
8   };
9
```

```

10     const _gamestates = {
11         chapitre_actuel : " chapitre 1",
12         niveau_actuel : niveau ${i} ',
13         choix_actuels : "a, b ou c ",
14         chapitre_précédent : " chapitre 1",
15         niveau_précédent : niveau ${i - 1} ' ,
16         choix_précédents : "a, b ou c ",
17         chapitre_cible : " chapitre 1",
18         niveau_cible : niveau ${i + 1} ' " ,
19         choix_cible : « a, b ou c
20     };
21
22     const _décisions = {
23         orchestration : 3,
24         adhésion : 0 vérification :
25         2 ,
26         conflit : 3,
27         choix : 4,
28         récupération : 5,
29         mauvaise conduite : 6 ,
30         anomalies : 1
31     };

```

Données simulées : les données de jeu (identifiants, états de jeu, décisions) sont générées pour chaque itération avec des valeurs spécifiques. Ces données simulent l'état d'un joueur dans un jeu.

```

1     Commençons , fin ;
2
3     essayer {
4         // Mesure du temps de démarrage pour l'ensemble du processus
5         début = Date . maintenant () ;

```

Mesure du temps : le temps nécessaire à chaque transaction est mesuré à l'aide de la
Les fonctions « Date.now() » sont exécutées avant et après l'envoi de la transaction.

```

1
2         // Générer un message à signer
3         message const = web3.utils.soliditySha3 ( Uuid stringify(_ids ) , Fichier JSON.
4         , JSON . stringify ( _gamestates ) stringify , Fichier JSON.
5         ( _décisions ));
6         const signature = wait web3 . eth . accounts . sign ( message ,
7         clé privée);

```

Signature du message : avant d'envoyer une transaction, le script génère un message à partir des données du jeu en utilisant « soliditySha3 » et le signe avec le compte privé de l'utilisateur clé. Cette signature garantit l'intégrité et l'authenticité des données.

```

1         // Envoi de la transaction avec signature incluse
2         const gasEstimate = attendre le contrat . méthodes . setGameState
3         ( Uuid _ids _décisions _gamestates , .
4         estimateGas ({ de : web3 . eth . defaultAccount }) ;

```

Estimation de gaz : le script estime la quantité de gaz nécessaire pour exécuter la fonction « setGameState » du contrat intelligent et ajoute une petite marge (10 000 unités de gaz).

```

1      const résultat = attendre le contrat . méthodes . setGameState ( UUID
      , _ids _décisions _gamestates , . send
2      ({ de : web3 . eth . defaultAccount , gas : Number (
      gasEstimate) + 10000, signature : signature.
      signature }));

```

Transaction envoyée : la transaction est envoyée au contrat intelligent, y compris le signature. Le résultat de la transaction est enregistré.

```

1      console . log ('Transaction ${i + 1} réussie .' , résultat )
      ;
2
3      // Vérifier la signature après la transaction
4      const isValid = wait verifySignature(UUID _ids, signature.signature);
      ,
      ,
      _gamestates , _decisions console . log ('
5      Résultat de la vérification de la signature : ${ isValid ?
      Valide      :      Invalide      }');

```

Vérification de la signature : une fois la transaction envoyée, le script vérifie que la signature est valide en la comparant avec l'adresse du compte qui a signé le message.

```

1      // Mesure du temps de fin pour l'ensemble du processus
2      fin = Date . maintenant () ;
3      console . log (" Durée totale de la transaction , Signature , "ms") ;
      et
      Vérification : " , fin - début
      ,

```

Mesure du temps : le temps nécessaire à chaque transaction est mesuré à l'aide de la fonction « Date.now() » sont exécutées avant et après l'envoi de la transaction.

```

1      } attraper (erreur) {
2      console . error (' Erreur dans la transaction ${i + 1} : '
      , erreur );
3      }
4      }
5  }

```

Gestion des erreurs : si une erreur survient pendant la transaction ou à toute autre étape, elle est capturée et enregistrée.

```

1
2 // Fonction de vérification de la signature de la blockchain
3 fonction asynchrone verifySignature ( UUID _ids , _gamestates , signature ) {
      _décisions
      ,
4      message const = web3 . utils . soliditySha3 ( UUID _ids )
      , JSON . chaîne de caractères (
      , JSON.stringify(_gamestates)
      , JSON . chaîne de caractères (
      _décisions ));
5      const downloadedAddress = wait web3.eth.accounts.recovery (
      message , signature );
6
7      renvoie l'adresse récupérée. toLowerCase() === web3 . eth.
      defaultAccount . toLowerCase() ;
8  }
9
10 créer des événements () ;

```

Fonction « VerifySignature » : cette fonction permet de vérifier qu'une signature spécifique correspond à l'adresse de l'utilisateur :

- Récréation du message : le message est recréé à l'identique de celui utilisé pour la signature.
- Récupération d'adresse : l'adresse qui a signé le message est récupérée à partir de la signature à l'aide de « web3.eth.accounts.recover ».
- Comparaison : l'adresse récupérée est comparée à l'adresse du compte par défaut pour vérifier la validité de la signature.
- Journalisation de sortie : le script redirige les sorties « console.log » et « console.error » vers un fichier journal (« nom.fichier.log ») ainsi que vers la sortie standard, vous permettant de suivre les événements dans un fichier.

Ce script est généralement utilisé pour automatiser l'envoi de transactions sur un contrat intelligent en simulant des événements de jeu. Il comprend également des mécanismes de vérification de la sécurité via des signatures numériques. Il s'agit d'un exemple d'intégration entre des systèmes hors chaîne (JavaScript, Web3) et des contrats intelligents sur une blockchain. Pour lancer le script, tapez la commande suivante dans un terminal VSCode :

1

```
$ node < nom_du_fichier >. js
```

6 Annexe

Structure de donnée utiliser pour modéliser un état de jeu dans un jeu narratif.

```

1 ID de structure {
2     chaîne player_id ;
3     chaîne game_id ;
4     chaîne reportée_player_id ;
5     chaîne reportée_game_id ;
6 }
7
8 structure État du jeu {
9     chaîne current_chapter ;
10    chaîne current_level ;
11    chaîne current_choices ;
12
13    chaîne chapitre_précédent ;
14    chaîne niveau_précédent ;
15    chaîne previous_choices ;
16
17    chaîne target_chapter ;
18    chaîne target_level ;
19    chaîne target_choices ;
20 }
21
22    enum Orchestration { Créé , Fermé , Inactif , En écrivant ,
23        En lisant , Avorter , Baisse }
24    enum Adhésion { Créé , Fermé , Inactif , Écriture , Lecture ,
25        Avorter , Baisse }
26    Vérification d'énumération { Créé , Fermé , Inactif , Écriture , Lecture
27        , Avorter , Baisse }
28    enum Conflit { Créé , Fermé , Inactif , Écriture , Lecture ,
29        Avorter , Baisse }
30    enum Choix { Créé , Fermé , Inactif , Écriture , Lecture ,
31        Avorter , Baisse }
32    enum Recovery { Créé , Fermé , Inactif , Écriture , Lecture ,
33        , Baisse }
34    Abandonner l'énumération Mauvais comportement { Créé , Fermé , Inactif , Écriture , Lecture
35        , Avorter , Baisse }
36    Anomalie {Aucun , Critique , Majeur , Mineure }
37
38 struct Décision {
39     Orchestration orchestration ;
40     Adhésion adhésion ;
41     Vérification vérification ;
42     Conflit conflit ;
43     Choix choix ;
44     Récupération récupération ;
45     Mauvaise conduite mauvaise conduite ;
46     Anomalie anomalie ;
47 }
48
49 struct InitializeGameState {
50     identifiants ;
51     État du jeu États du jeu ;
52     Décisions décisions ;
53 }

```

Liste 3 : Structure de donnée

Le smart-contract utilis'e pour la sauvegarde d'un 'etat de jeu.

```

1 // SPDX - Licence - Identifiant : MIT
2 pragma solidité ^0.8.0;
3
4 contrat GameDataStorage {
5     ID de structure {
6         chaîne player_id ;
7         chaîne game_id ;
8         chaîne reportée_player_id ;
9         chaîne reportée_game_id ;
10    }
11
12    structure État du jeu {
13        chaîne current_chapter ;
14        chaîne current_level ;
15        chaîne current_choices ;
16
17        chaîne chapitre_précédent ;
18        chaîne niveau_précédent ;
19        chaîne previous_choices ;
20
21        chaîne target_chapter ;
22        chaîne target_level ;
23        chaîne target_choices ;
24    }
25
26    enum Orchestration { Créé , Fermé , Inactif , En écrivant ,
27        En lisant , Avorter , Baisse }
28    enum Adhésion { Créé , Fermé , Inactif , Écriture , Lecture ,
29        Avorter , Baisse }
30    Vérification d'énumération { Créé , Fermé , Inactif , Écriture , Lecture ,
31        Avorter , Baisse }
32    enum Conflit { Créé , Fermé , Inactif , Écriture , Lecture ,
33        Avorter , Baisse }
34    enum Choix { Créé , Fermé , Inactif , Écriture , Lecture ,
35        Avorter , Baisse }
36    enum Recovery { Créé , Fermé , Inactif , Écriture , Lecture ,
37        Baisse }
38    Abandonner l'énumération Mauvais comportement { Créé , Fermé , Inactif , Écriture , Lecture ,
39        Avorter , Baisse }
40    Anomalie {Aucun , Critique , Majeur , Mineure }
41
42    structure Décision {
43        Orchestration orchestration ;
44        Adhésion adhésion ;
45        Vérification vérification ;
46        Conflit conflit ;
47        Choix choix ;
48        Récupération récupération ;
49        Mauvaise conduite mauvaise conduite ;
50        Anomalie anomalie ;
51    }
52

```



```

46     structure InitializeGameState {
47         identifiants ;
48         État du jeu États du jeu ;
49         Décisions décisions ;
50     }
51
52     // Mappage pour stocker l'initialisation de l'état du jeu par player_id
53     mappage ( chaîne => InitializeGameState ) public
        initialiser les états du jeu ;
54
55     fonction setGameState (
56         chaîne mémoire player_id ,
57         Mémoire d'identification _ids ,
58         Mémoire GameState _gameStates ,
59         Mémoire de décision _décisions
60     ) publique {
61         Initialiser la mémoire GameState newGameState =
            Initialiser l'état du jeu (
62             _ids ,
63             _États du jeu ,
64             _décisions
65         );
66         initializeGameStates [ player_id ] = newGameState ;
67     }
68
69     fonction setOrchestrationState ( chaîne mémoire player_id ,
        État d'orchestration ) public {
70         initializeGameStates [ player_id ]. décisions . orchestration =
            État ;
71     }
72
73     fonction setMembershipState ( chaîne mémoire player_id , Membership
        état ) public {
74         initializeGameStates [ player_id ]. décisions . membership =
            État ;
75     }
76
77     fonction setVerificationState ( chaîne mémoire player_id ,
        État de vérification ) public {
78         initializeGameStates [ player_id ]. décisions . verification =
            État ;
79     }
80
81     fonction setConflictState ( chaîne mémoire player_id , état ) public {
674         Conflit
82         initializeGameStates [ player_id ]. décisions . conflict = état ;
83     }
84
85     fonction setChoiceState ( chaîne mémoire player_id , public {
674         État de choix )
86         initializeGameStates [ player_id ]. décisions . choice = état ;
87     }
88
89     fonction setRecoveryState ( chaîne mémoire player_id , Récupération
        état ) public {
90         initializeGameStates [ player_id ]. décisions . recovery = état ;
91     }

```

```

92
93     fonction setMisbehaviourState ( chaîne mémoire player_id ,
          État de mauvaise conduite ) public {
94         initializeGameStates [ player_id ]. decisions . misbehaviour =
          État ;
95     }
96
97     fonction setAnomalieState ( chaîne mémoire player_id , état ) public {           Anomalies
98         initializeGameStates [ player_id ]. decisions . anomalie = état ;
99     }
100 }

```

Liste 4 : Smart contract en Solidity

```

1 const { Web3 } = require ( ' web3 ' );
2 const fs = require ( ' fs ' );
3 const util = require ( ' util ' );
4
5 // Initialiser Web3
6 const web3 = new Web3 ( ' adresse RPC ici ' );
7
8 // La clé privée de votre compte (assurez-vous de la garder en sécurité)
9 const privateKey = ' clé privée ici ' ;
web3 . eth . accounts . privateKeyToAccount ( privateKey );
11 web3 . eth . comptes . wallet . add ( compte );
12 web3 . eth . defaultAccount = compte . adresse ;
13
14 // ABI de votre contrat intelligent
15 const contractABI = [ ' ABI a coller ici ici ' ];
16
17 // L'adresse de votre contrat déployé
18 const contractAddress = ' adresse du contrat a mettre ici ' ;
19
20 // Créer une instance de contrat
21 const contract = nouveau web3 . eth . Contract ( contractABI , contractAdresse );
22
23 const logFile = fs . createWriteStream ( ' transactions_01 . log ' , { flags
    : ' un ' } );
24 const logStdout = processus . stdout ;
25
26 console . log = fonction () {
27     fichierjournal.write(util.format.apply(null, arguments) + '\n');
28     logStdout.write(util.format.apply(null, arguments) + '\n');
29 }
30
31 console . erreur = fonction () {
32     fichierjournal.write(util.format.apply(null, arguments) + '\n');
33     logStdout.write(util.format.apply(null, arguments) + '\n');
34 }
35
36 fonction asynchrone createEvents() {
37     pour ( soit i = 0; i < 100; i ++ ) { ,
38         const UUID = exemple - uuid - playerId - ${i} ;
39         const _ids = { ,
40             identifiant_joueur : ID joueur - ${i} ,

```

```

41         game_id : 'OMFG '
42         signalé_player_id : 'no_id_player',
43         report_game_id : 'OMFG '
44     };
45
46     const _gamestates = {
47         current_chapter : ' chapitre 1',
48         niveau_actuel :          niveau ${i} ',
49         current_choices : 'à propos de ',
50         chapitre_précédent :          ' chapitre 1 ',
51         niveau_précédent :          niveau ${i - 1} '
52         ,
53         choix_précédents : 'a chapitre_cible : , b ou c',
54         niveau_cible : choix_cibles ; ' chapitre 1 ',
55         'a          niveau ${i + 1} '
56         , b ou c
57     };
58
59     const _décisions = {
60         orchestration : 3,
61         adhésion : 0 vérification : ,
62         2
63         ,
64         conflit : 3,
65         choix : 4,
66         récupération : 5,
67         mauvaise conduite : 6
68         ,
69         anomalies : 1
70     };
71
72     Commençons , fin ;
73
74     essayer {
75         // Mesure du temps de démarrage pour l'ensemble du processus
76         début = Date . maintenant () ;
77
78         // Envoi de la transaction sans signature incluse
79         const gasEstimate = attendre le contrat . méthodes . setGameState
80         ( Uuid _décisions , _gamestates , . estimateGas
81         ({ de : web3 . eth . defaultAccount }) );
82
83         const résultat = attendre le contrat . méthodes . setGameState ( Uuid
84         , _ids , _gamestates , . send ({ de : _décisions )
85         web3 . eth . defaultAccount , gas : Number (
86         gasEstimated ) + 10000 }) ;
87
88         console . log ('Transaction ${i + 1} réussie .' , résultat )
89         ;
90
91         // Mesure du temps de fin pour l'ensemble du processus
92         fin = Date . maintenant () ;
93         console . log ('EventStorageTime' , fin - début ) ;
94
95     } attraper (erreur) {
96         console . error (' Erreur dans la transaction ${i + 1} : ' , erreur );
97     }
98 }
99

```

```
94 créer des événements () ;
```

Liste 5 : code Node JS