# Recap Ethereum Design Patterns

Blockchain-Based Systems Engineering

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

## Solidity Idioms

- **Programming idioms** are language-specific patterns for recurring programming problems.
- Idioms are on a lower abstraction level than **design patterns** which are template solutions for recurring software engineering problems.
- OpenZeppelin is a library that automates operations and delivers reusable, secure, tested and community-audited code. Most of the critical building blocks that are needed for a contract are already pre-programmed in it, so users should utilize the existing library instead of writing their own code. In this section, we will go through the most prevalent idioms of Solidity smart contract programming through the OpenZeppelin library.
- *SafeMath*[1] is library that validates if an arithmetic operation would cause an integer overflow/underflow.
  - Until Solidity *version 0.8*, it had to be manually included and utilized by the smart contract developer
  - Since *version 0.8*, it is implemented on the language level

```
// Java idiom for generating random number.
Random rand = new Random();
int diceRoll() {
  return rand.nextInt(6) + 1;
}
```

A **Solidity idiom** is a practice-proven code **pattern** for a **recurring coding problem.**

[1] https://docs.openzeppelin.com/contracts/2.x/api/math
https://github.com/ethereum/wiki/wiki/Useful-%C3%90app-Patterns
*Alexander Hefele, "A Conceptual Model for Ethereum Blockchain Analytics", Master's Thesis, Technical University Munich, 2019.*

## Access Restriction Idiom

**Description**

- Each contract deployed on the main network is publicly accessible.
- Since all external and public functions can be called by anyone, third parties might execute a function on a contract they should not be allowed to.
- Misconfigured access restrictions led to the largest Ethereum thefts so far[1].

**Participants**

- An entity that calls a publicly accessible function in a smart contract.
- A smart contract which is called by a transaction or a message.

**Applicability**

- To protect contract functions from unauthorized calls.
- Examples for such functions: selfdestruct(), mint()

[1] *https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7, accessed on 02.06.2019*

## Access Restriction Idiom (cont.)

**Solution**

Restrict access of other accounts to execute functions and to modify the state of a contract. In Solidity, access restriction can be achieved by implementing proper **function modifiers** that check if the caller is allowed to execute the actual function logic. To make the contract code more maintainable, the authorization logic is usually put in a separate contract.

### Ownable Contract

```
contract Ownable {
        address public owner;

        constructor() public {
                owner = msg.sender;
        }

        /**
        * @dev Throws if called by any account other than the owner.
        */
        modifier onlyOwner() {
                require(msg.sender == owner);
                _;
        }
/...
```

### Contract which implements Ownable

```
/...
        /**
        * @dev Allows the current owner to transfer control
        * @param newOwner The address to transfer ownership to.
        */
        function transferOwnership(address newOwner) public onlyOwner {
                        require(newOwner != address(0));
                        owner = newOwner;
        }

        /**
        * @dev Allows the current owner to relinquish control */
        function renounceOwnership() public onlyOwner {
                        owner = address(0);
        }
}
```

## Smart Contract Design Patterns

Designing decentralized applications on the basis of a blockchain infrastructure is a rather new area in software engineering. Similar to traditional software engineering, there are recurring problems that are shared across a large set of smart contracts.

**Design patterns** are **template solutions** for **recurring design problems**.

Design patterns are specifically important for smart contract development:

- Determinism by design makes use of external data and random numbers challenging.
- The code of a deployed contract is immutable ➔ Contracts cannot be updated.
- In most cases, financial value is at risk.
- Transaction finality ➔ Stolen money is gone forever.
- Availability of source and bytecode makes it easier for attackers to find potential vulnerabilities.

https://github.com/ethereum/wiki/wiki/Useful-%C3%90app-Patterns

## Oracle Pattern

**Problem Description**

**Smart contracts can’t access** any **data** from **outside** the **blockchain** on their own. There are no HTTP or similar network methods implemented to call or access external services. This is on purpose to **prevent non-deterministic behavior** once a function is called (there are also no functions to generate random values).

**Participants**

- A smart contract which **requires data** from external sources
- An external party that is willing to **provide data** from external sources via a **separate smart contract**

**Applicability**

Any scenario in which a smart contract **relies on external data for computation** of future states.

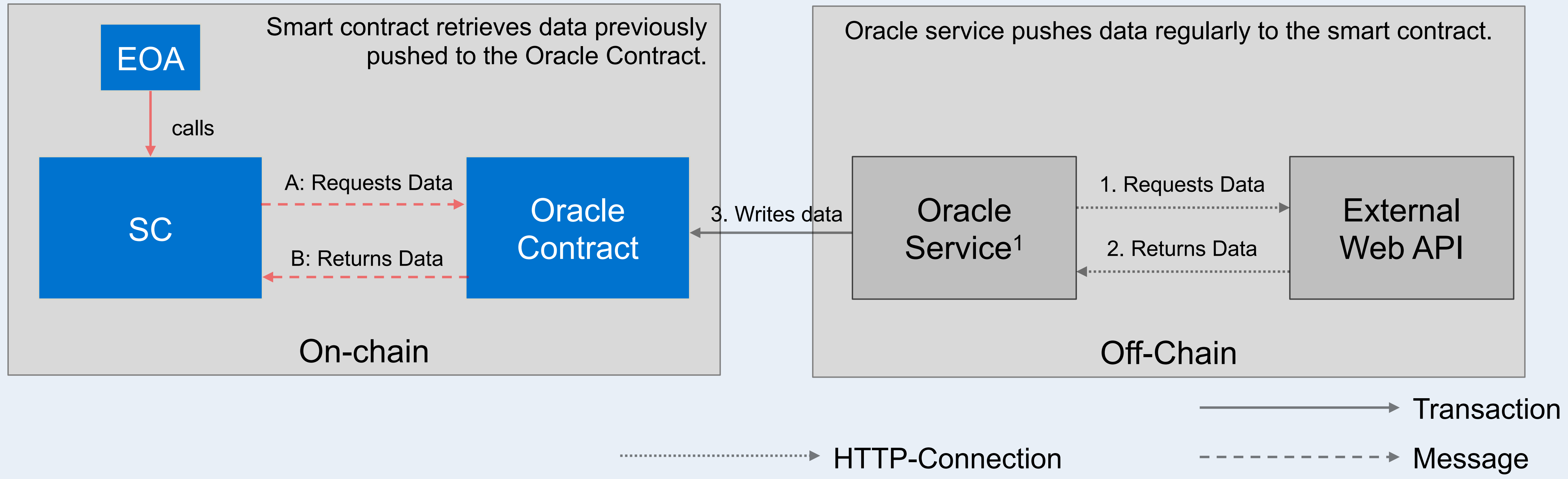
Check out https://data.chain.link/

## Synchronous Oracle

**Solution**

Currently, the **only way** to write smart contracts **using external data** (e.g. weather data, traffic data etc.) is to **use oracles**. Oracles are basically third-party services that collect data from web services and write the data via a special smart contract to the blockchain. Other smart contracts can now call the Oracle contract to get the data. We differentiate between **synchronous** and **asynchronous** Oracles.

**Synchronous Oracle**



¹The Oracle service also needs an EOA (externally owned account) to write to the Smart Contract.

## Synchronous Oracle Contract Example

A simple Oracle contract for the current market prices of Ethereum and Bitcoin.

```
contract SimpleOracle {
   address public controller;
   enum Coins {Ethereum, Bitcoin}
   Coins public coins;
   mapping (uint => uint) public prices;

   constructor() public {
      controller = msg.sender;
   }

   modifier isController {
      require(msg.sender == controller);
      _;
   }

   function update(uint coin, uint price) public isController {
      if(coin != uint(coins.Ethereum) && coin != uint(coins.Bitcoin)) {
         revert();
      }
      prices[coin] = price;
   }

   function getPrice(uint coin) public view returns(uint price) {
      return prices[coin];
   }
}
```

The controller of the Oracle contract

The storage that represents the market data

Controller is set by contract deployment

An update function that lets the controller of the contract set the current market prices

A publicly accessible view function that returns the current price of a particular coin.

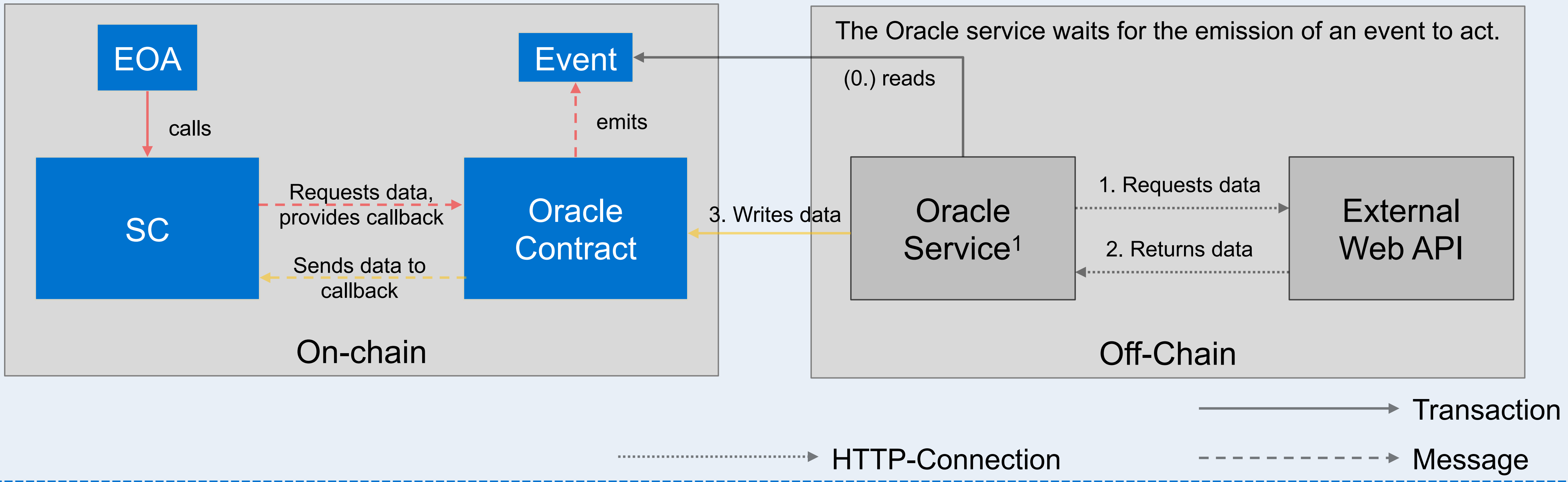
## Asynchronous Oracle

Inversion of Control

**Solution**

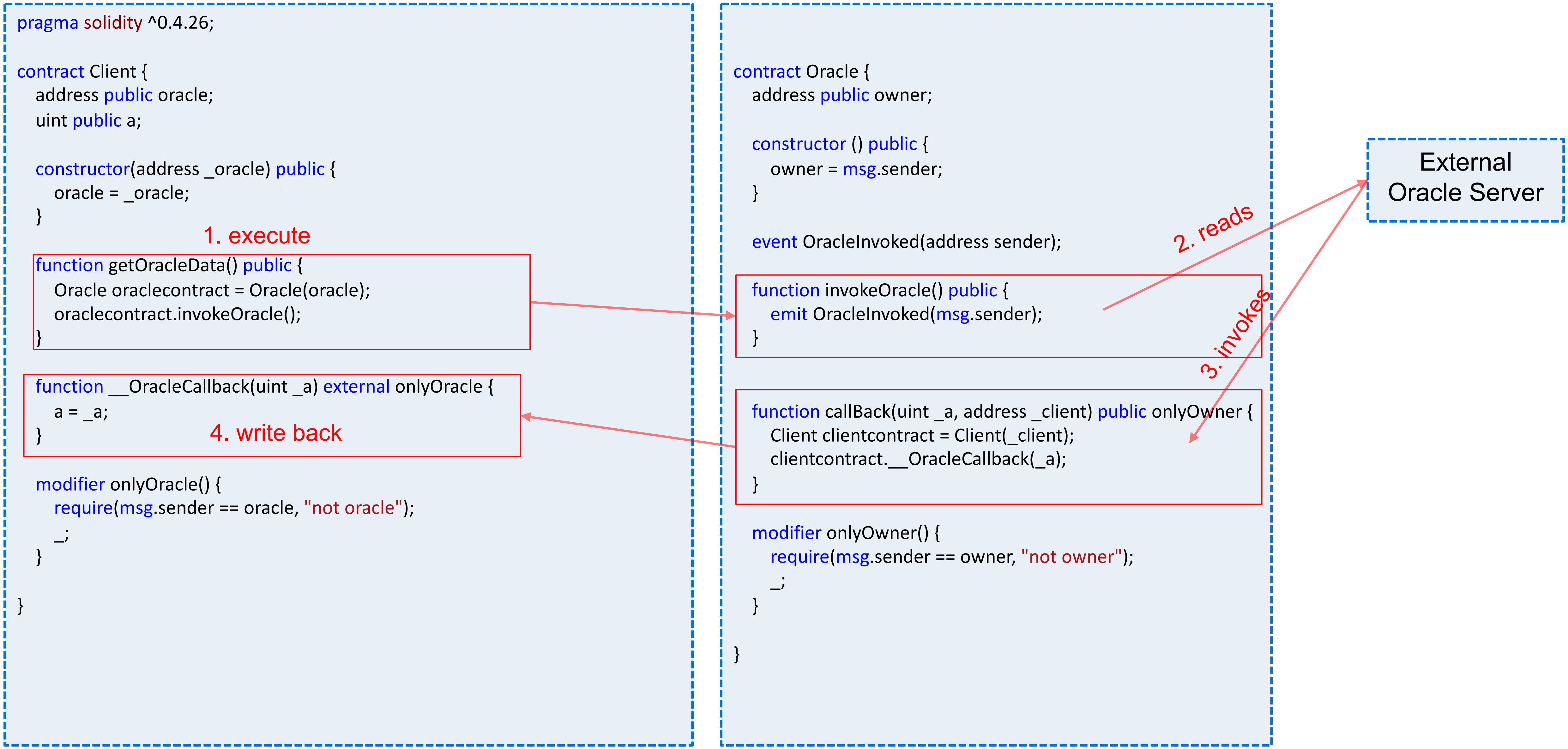
In contrast to the synchronous Oracle (in which data is periodically pushed to the Oracle contract and retrieved by the smart contract), the asynchronous Oracle acts as a proxy for the Oracle service to send fresh data to the original smart contract.

**Asynchronous Oracle**



[1] *The Oracle service also needs an EOA (externally owned account) to write to the Smart Contract.*

# Asynchronous Oracle Contract Example



```
pragma solidity ^0.4.26;

contract Client {
   address public oracle;
   uint public a;

   constructor(address _oracle) public {
      oracle = _oracle;
   }

   function getOracleData() public {
      Oracle oraclecontract = Oracle(oracle);
      oraclecontract.invokeOracle();
   }

   function __OracleCallback(uint _a) external onlyOracle {
      a = _a;
   }

   modifier onlyOracle() {
      require(msg.sender == oracle, "not oracle");
      _;
   }

}
```

```
contract Oracle {
   address public owner;

   constructor () public {
      owner = msg.sender;
   }

   event OracleInvoked(address sender);

   function invokeOracle() public {
      emit OracleInvoked(msg.sender);
   }

   function callBack(uint _a, address _client) public onlyOwner {
      Client clientcontract = Client(_client);
      clientcontract.__OracleCallback(_a);
   }

   modifier onlyOwner() {
      require(msg.sender == owner, "not owner");
      _;
   }

}
```

## Limit Gas Usage in Asynchronous Oracle Contract

- In case of the asynchronous oracle, the oracle server sends a **transaction** to the oracle contract which **invokes** a **message** to the client contract.
- In our case, **all gas** from the original transaction **is forwarded** to the message.
- The Client smart contract could consume this gas for malicious purposes.
- Therefore, it is advisable to **limit** the **forwarded gas**. The limitation is introduced with an additional parameter. The parameter _gas defines the to-be forwarded amount of gas.

  clientcontract.__OracleCallback(**gas _gas**)(_a);

- **Be careful**: Low gas can make transactions fail!

```
contract Oracle {
   address public owner;

   constructor () public {
      owner = msg.sender;
   }

   event OracleInvoked(address sender);

   function invokeOracle() public {
      emit OracleInvoked(msg.sender);
   }


   function callBack(uint _a, address _client, uint _gas) public onlyOwner {
      Client clientcontract = Client(_client);
      clientcontract.__OracleCallback(gas _gas)(_a);
   }

   modifier onlyOwner() {
      require(msg.sender == owner, "not owner");
      _;
   }

}
```