Smart Contracts

am Beispiel Ethereum

Inhalt

1 Was ist ein Smart Contract?

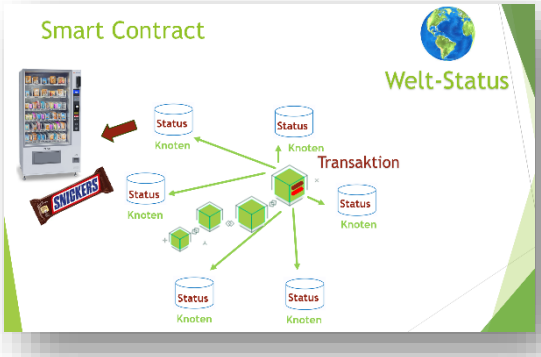
1.1 Contracts von heute und morgen

1.2 Smart Contract am Beispiel eines Verkaufsautomaten

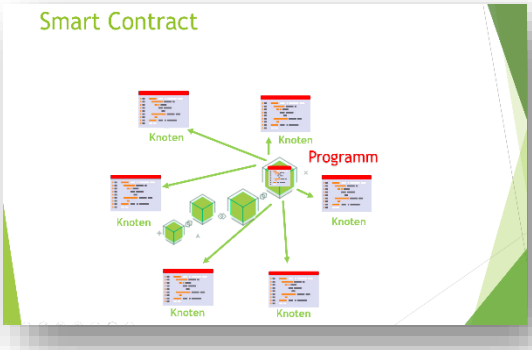
Eine einfache Metapher für einen Smart Contract ist ein Verkaufsautomat, dessen Funktionsweise mit einem Smart Contract vergleichbar ist: Konkrete Eingaben garantieren vorab festgelegte Leistungen.

Der Verkaufsautomat gibt das gewünschte Produkt erst dann aus, wenn alle Anforderungen erfüllt sind. Wenn Sie beispielsweise kein Produkt auswählen oder nicht genug Geld einwerfen, gibt der Automat kein Produkt aus.

Der weltweite Abgleich einer verteilten Datenbank mit unbekannten und möglichweisen korrupten Teilnehmern ist eine große Herausforderung.

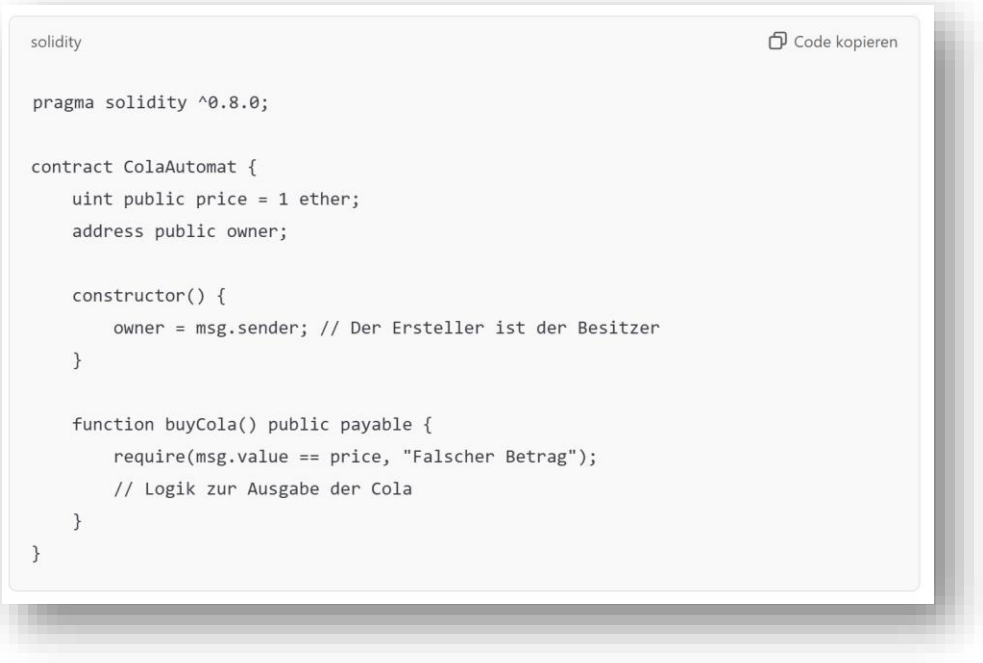


Die Statusänderungen eines Verkaufsvorgangs werden auf einer öffentlichen Blockchain (z.B. Ethereum) in Form von Transaktionen gespeichert und weltweit auf allen Knoten umgesetzt. Jeder kann zeitnah die Übertragung von Vermögenswerten und weitere damit verbundenen Informationen nachvollziehen.



Auf der öffentlichen Blockchain können mit Transaktionen nicht nur Coins transferiert werden, sondern auch kleine Programme mit „Wenn-Dann-Anweisungen“, sogenannte „Smart Contracts“ an die Knoten ausgeliefert werden. Die Smart Contracts werden dann über Blockchain-Transaktionen angesteuert.

1.3 Smart Contract Beispiel



solidity Code kopieren

```
pragma solidity ^0.8.0;

contract ColaAutomat {
    uint public price = 1 ether;
    address public owner;

    constructor() {
        owner = msg.sender; // Der Ersteller ist der Besitzer
    }

    function buyCola() public payable {
        require(msg.value == price, "Falscher Betrag");
        // Logik zur Ausgabe der Cola
    }
}
```

1.4 Definition des Begriffs „Smart Contract“

1. Smart Contracts sind Computerprogramme,
   die auf der Blockchain gespeichert sind
   und der „Wenn dies, dann das“-Logik folgen.
2. Smart Contracts werden garantiert nach den Regeln ausgeführt,
   die durch ihren Code definiert sind.
3. Smart Contracts können nach der Erstellung nicht mehr geändert werden.

Man kann sich Smart Contracts wie blockchainbasierte digitale Verträge vorstellen, die keiner menschlichen Kontrolle bedürfen, sondern auf festgelegten und manipulationssicheren Regeln basieren.

1.5 Vorteile von Smart Contracts

- **Transparenz**:
  Alle Regeln sind offen einsehbar.
- **Manipulationssicherheit**:
  Der Smart Contract kann nicht manipuliert oder geändert werden, weil er auf der Blockchain liegt.
- **Dezentralität**:
  Kein Unternehmen oder Dritter muss das System überwachen, da der Smart Contract automatisch arbeitet.

2 Cola-Automat als Smart Contract

2.1 Aufgabe

Erstellen Sie einen **Smart Contract** in der Programmiersprache **Solidity** auf dem Testnetz **Sepolia.**

Der **Smart Contract** soll es ermöglichen, einen **Cola-Token** zu einem Preis von ca. **1,- €** mit der Währung **Ether** zu bezahlen.

Die Umsetzung dieser Aufgabenstellung wird im Folgenden erläutert.

2.2 Programmiersprache Solidity

Solidity ist eine **objektorientierte, hochspezialisierte Programmiersprache**, die primär für die Entwicklung von **Smart Contracts** auf der **Ethereum-Blockchain** und ähnlichen Plattformen konzipiert wurde. Sie wurde von **Gavin Wood**, einem der Mitbegründer von Ethereum, entwickelt und ist seit ihrer Einführung die Hauptsprache für die Erstellung dezentraler Anwendungen (DApps).

Eigenschaften von Solidity

1. **Smart Contract**: Solidity wurde speziell entwickelt, um Verträge auszuführen, die die Logik für dezentrale Anwendungen beinhalten.
2. **Syntax**: Die Syntax von Solidity ist an bekannte Sprachen wie **JavaScript**, **Python** und **C++** angelehnt. Dies erleichtert Entwicklern mit Erfahrung in diesen Sprachen den Einstieg.
3. **Turing-vollständig**: Solidity ermöglicht das Schreiben von komplexen und vielseitigen Programmen.
4. **Plattform**: Solidity wird hauptsächlich für Ethereum verwendet, ist aber auch auf anderen Ethereum-kompatiblen Blockchains (z. B. Binance Smart Chain, Polygon) einsetzbar.

Vorteile von Solidity

- Breite Unterstützung durch die Ethereum-Community.
- Starke Integration mit Blockchain-Technologien.
- Leistungsstarke Tools und Frameworks verfügbar.

Herausforderungen bei der Solidity-Programmierung

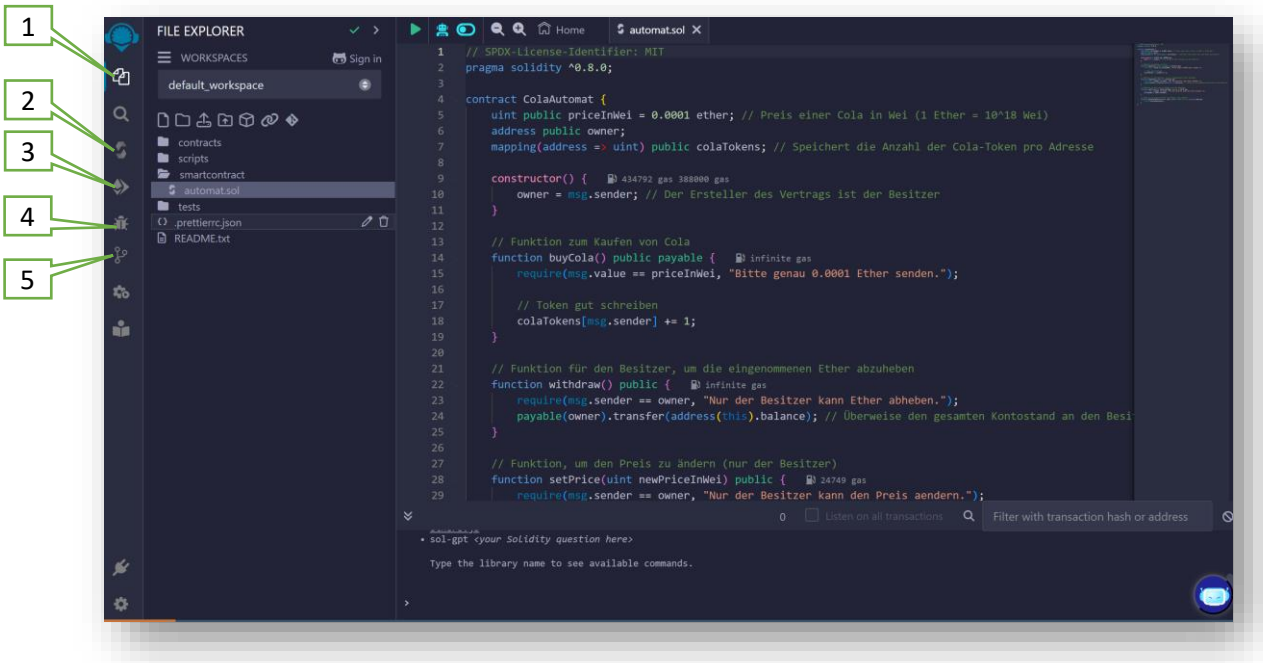
1. **Sicherheitsrisiken**:
   Fehler im Code können erhebliche finanzielle Verluste verursachen.
2. **Komplexität**:
   Entwickler müssen sowohl Blockchain- als auch Solidity-Kenntnisse haben.
3. **Gas-Kosten**:
   Ineffizienter Code kann teure Transaktionen verursachen.

2.3 Web-basierte-Entwicklungsumgebung „Remix“

Remix ist eine **Web-basierte Entwicklungsumgebung (IDE)**, die speziell für die Erstellung, das Testen und das Bereitstellen von **Smart Contracts** auf der **Ethereum-Blockchain** entwickelt wurde. Es ermöglicht Entwicklern, schnell und einfach mit **Solidity**, der Programmiersprache für Ethereum Smart Contracts, zu arbeiten. Remix läuft vollständig im Browser und ist damit **plattformunabhängig.** Somit muss keine Software installiert werden und man kann sofort loslegen.

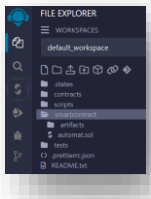
Die Plattform ist zu erreichen unter: https://remix.ethereum.org/

Der Browser merkt sich automatisch (auch ohne Account) alle Einstellungen, Ordnerstrukturen und Programme über Cookies.



1. **Code-Editor**: Ein leistungsstarker Texteditor, der speziell für Solidity entwickelt wurde. Er bietet Syntax-Highlighting, Autovervollständigung und andere Funktionen, die das Schreiben von Code erleichtern.
2. **Solidity-Compiler**: Remix enthält einen integrierten Solidity-Compiler, der den geschriebenen Code in Bytecode übersetzt, den die Ethereum-Blockchain ausführen kann.
3. **Deployment**: Sobald der Smart Contract fertig ist, kannst du ihn direkt von der Remix-Oberfläche aus auf das Ethereum-Netzwerk hochladen und in Betrieb nehmen.
4. **Testing & Debugging**: Remix ermöglicht es, Smart Contracts auf verschiedenen Testnetzwerken (wie Rinkeby oder Ganache) zu testen, bevor sie auf das Hauptnetzwerk (Mainnet) hochgeladen werden. Es bietet auch eine detaillierte Debugging-Ansicht.
5. **Git-Button:** Remix ermöglicht die Integration von Git, einem Versionskontrollsystem, direkt in die Remix-IDE. Diese Funktion ist besonders nützlich, um Smart-Contract-Projekte zu verwalten, Änderungen nachzuverfolgen und Projekte mit anderen zu teilen.

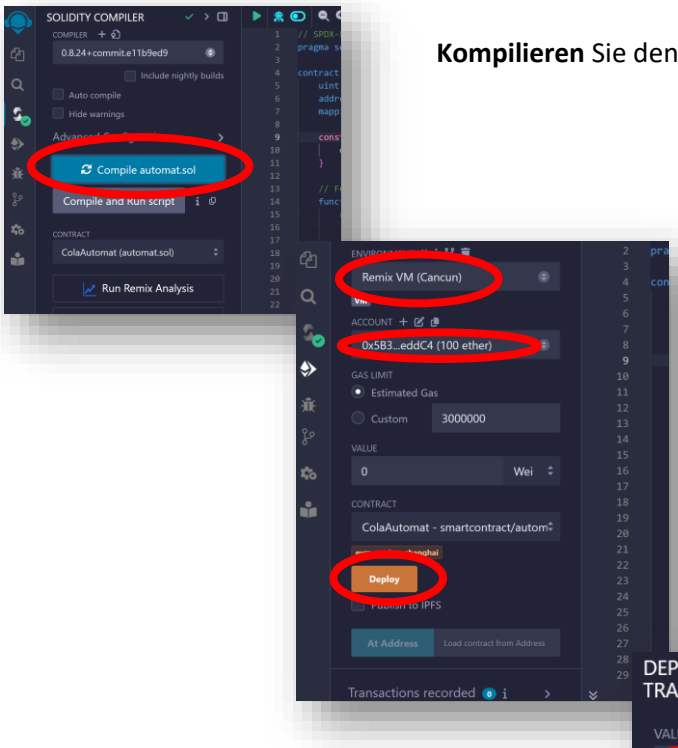
2.4 Solidity-Sourcecode



Erstellen Sie in Remix einen Ordner „**smartcontract**“ und darin die Datei „**automat.sol**“.
Übertragen Sie den folgenden Sourcecode:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ColaAutomat {
    uint public priceInWei = 0.0001 ether; // Preis einer Cola in Wei (1 Ether = 10^18 Wei)
    address public owner;
    mapping(address => uint) internal colaTokens; // Speichert die Anzahl der Cola-Token pro Adresse

    constructor() {
        owner = msg.sender; // Der Ersteller des Vertrags ist der Besitzer
    }

    // Funktion zum Kaufen von Cola
    function buyCola() public payable {
        require(msg.value == priceInWei, "Bitte genau 0.0001 Ether senden.");

        // Token gut schreiben
        colaTokens[msg.sender] += 1;
    }

    // Funktion für den Besitzer, um die eingenommenen Ether abzuheben
    function withdraw() public {
        require(msg.sender == owner, "Nur der Besitzer kann Ether abheben.");
        payable(owner).transfer(address(this).balance); // Überweise den gesamten Kontostand an den Besitzer
    }

    // Funktion, um den Preis zu ändern (nur der Besitzer)
    function setPrice(uint newPriceInWei) public {
        require(msg.sender == owner, "Nur der Besitzer kann den Preis aendern.");
        priceInWei = newPriceInWei;
    }

    // Funktion zum Überprüfen des Cola-Tokens einer Adresse
    function getColaTokens(address user) public view returns (uint) {
        return colaTokens[user];
    }
}
```

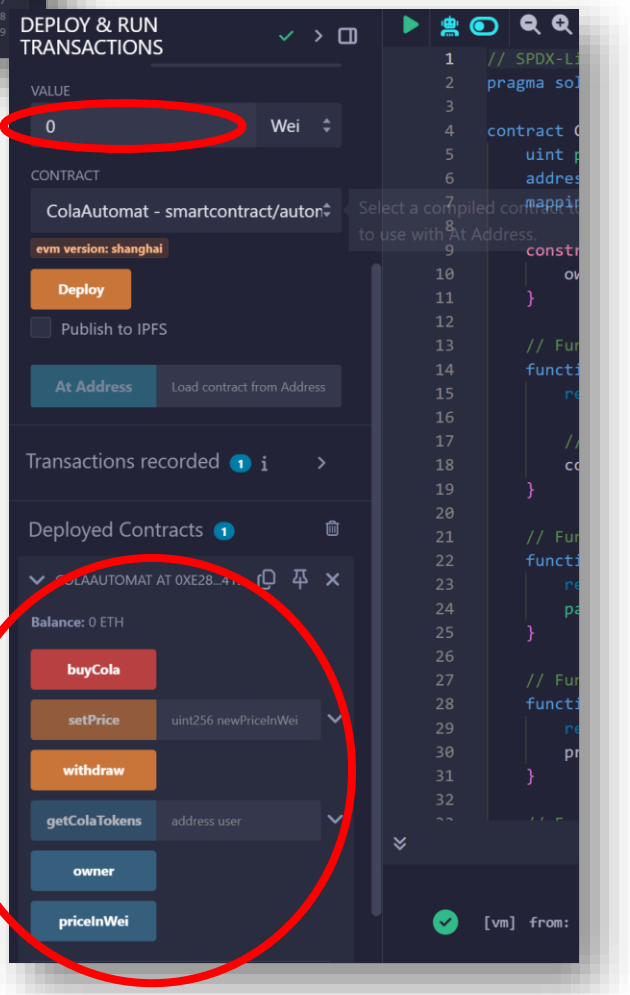
2.5 Vertrag kompilieren, veröffentlichen, ansprechen



Kompilieren Sie den Sourcecode!

Veröffentlichen Sie den Smartcontract anschließend auf dem Testnetz „**Remix VM**“.

Über das Auswahlfeld „**Account**“ können Sie zwischen mehrere Konten wählen, die jeweils mit 100 Ether gefüllt sind.



Über das Eingabefeld „Value“ kann man einer Transaktion einen **Wert** mitgeben. Dieser wird benutzt, um Ether zu transferieren.

Remix analysiert den Sourcecode des Vertrages, erkennt alle ansprechbaren Elemente (Funktionen, Variablen) und generiert eine **Schnittstelle**, um entsprechende Transaktionen bzw. Calls an den Vertrag zu senden.

2.6 Transaktion vs. Call

Bei Ethereum besteht der Hauptunterschied zwischen einer **Transaktion** und einem **Call** darin, ob eine Änderung am Zustand der Blockchain vorgenommen wird oder nicht:

Transaction

- **Definition:** Eine Transaktion ist eine signierte Nachricht, die von einem Ethereum-Konto gesendet wird, um eine Aktion auf der Blockchain auszuführen. Sie führt zu einer Änderung des Zustands der Blockchain. Eine Transaktion wird in einem Block aufgenommen und durch die Blockchain validiert.
- **Merkmale:**
  - **Zustandsänderung:** Führt zur Aktualisierung des globalen Zustands, z. B. Überweisungen von Ether, Interaktionen mit Smart Contracts oder das Deployment eines neuen Contracts.
  - **Kosten:** Transaktionen verbrauchen Gas (bezahlt in Ether), da sie Arbeit von den Minern/Validatoren erfordern.
  - **Persistenz:** Das Ergebnis einer Transaktion wird in der Blockchain gespeichert und ist dauerhaft.
  - **Beispiel:** Übertragen von Ether von einer Adresse zu einer anderen.

Call

- **Definition:** Ein Call ist eine lesende Abfrage, die an einen Smart Contract gerichtet wird, ohne den Zustand der Blockchain zu ändern. Calls werden nicht in die Blockchain aufgenommen, da sie keine Transaktionen sind.
- **Merkmale:**
  - **Keine Zustandsänderung:** Call-Operationen führen keine Änderungen am Zustand der Blockchain durch (sind also „read-only“).
  - **Kostenlos:** Calls verbrauchen kein Gas, da sie lokal von einem Ethereum-Knoten (Node) ausgeführt werden und keine Miner- oder Validatoren-Ressourcen benötigen.
  - **Lokal:** Calls werden lokal ausgeführt und beeinflussen den globalen Zustand nicht.
  - **Beispiel:** Abfrage eines gespeicherten Werts in einem Smart Contract.

Zusammenfassung

Merkmal	Transaktion	Call
Zustandsänderung	Ja	Nein
Gasverbrauch	Ja	Nein
Speicherung in Blockchain	Ja	Nein
Verwendungszweck	Schreiben/Ausführen von Änderungen	Lesen von Daten

2.7 Aufgaben

Aufgabe 1:

a. Ermitteln Sie über die Schnittstelle in Remix den „Preis in Wei“.
Tragen Sie diesen Preis bei Remix als „Value“ ein und kaufen Sie einen Cola-Token!

b. Erläutern Sie den Zusammenhang von Ether, GWei und Wei.

c. Setzen Sie den Preis (über „SetPrice“) so, dass der Cola-Token ca. 1,- € kostet.

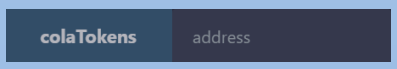
Preis in Wei: _____

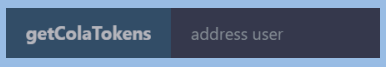
d. In Solidity wird der Betrag 0,0001 Ether nicht als Kommazahl, sondern als sehr großer Betrag in der Einheit Wei gespeichert. Recherchieren Sie die Begründung!

Aufgabe 2:

Analysieren Sie den Programmcode. Nutzen Sie hierfür die Solidity-Befehlsübersicht aus dem Anhang.

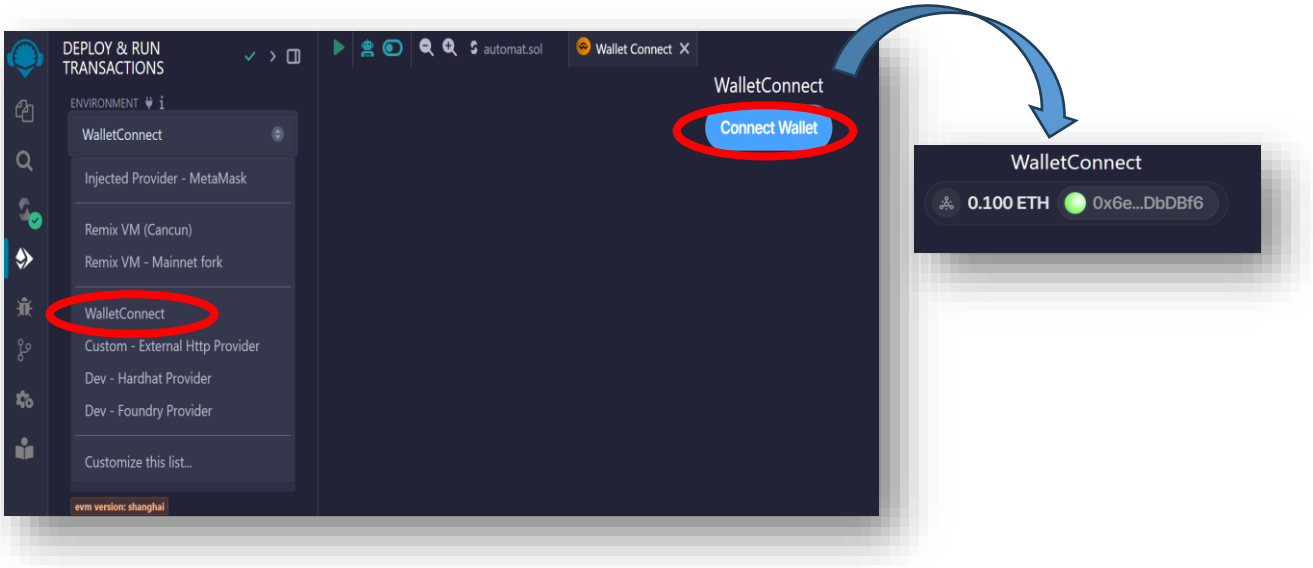
Erläutern Sie die folgenden Programmfunktionen in Stichpunkten.
Was bedeuten die Farben der Buttons? (Tipp: „MousOver“ nutzen)
Bereiten Sie eine entsprechende Remix-Kurzpräsentation vor!



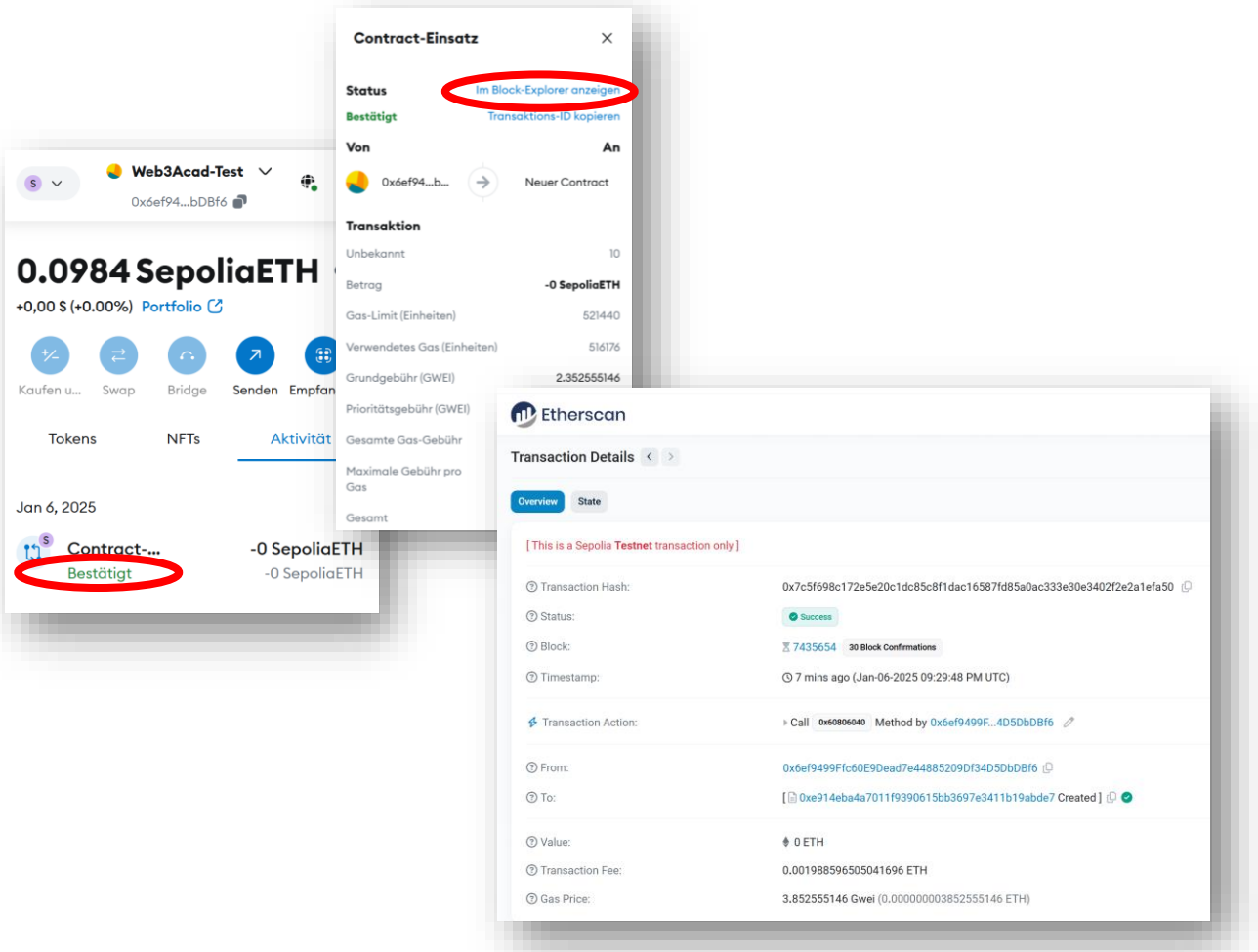


2.12 Vertrag auf Sepolia veröffentlichen

Damit wir den Vertrag auf dem Sepolia-Testnetz veröffentlichen können, müssen Remix mit dem Testnetz verbunden werden. Wie man in der „Environment-Auswahlbox“ sieht, wird das Sepolia-Testnetz leider nicht direkt angeboten. Daher müssen wir erst unsere MetaMask-Wallet mit Remix verbinden und dann in MetaMask das Sepolia-Netz auswählen.



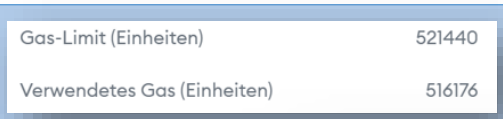
Veröffentlichen Sie nun den Vertrag auf dem Sepolia-Testnetz und analysieren Sie das Ergebnis in der MetaMask-Wallet und im Block-Explorer.



2.13 Aufgaben

Aufgabe 3:

Warum ist das Gas-Limit bei der Ausführung eines Ethereum-Smart Contracts wichtig?



3 Solidity Befehlsübersicht

Kategorie	Befehl/Schlüsselwort	Beschreibung
Versionsangabe	pragma solidity ^x.x.x;	Gibt die Solidity-Version an, die der Compiler verwenden soll.
Lizenzdeklaration	// SPDX-License-Identifier	Lizenztyp für den Code (z. B. MIT, GPL).
Vertragsdeklaration	contract ContractName {}	Definiert einen Smart Contract.
Variablen	uint, int, bool, address	Definiert Variablen. Unterstützt primitive Typen wie Ganzzahlen, Adressen usw.
	mapping(KeyType => ValueType)	Key-Value-Speicherstruktur.
	struct StructName {}	Definiert benutzerdefinierte Typen.
Zustandsvariablen	storage, memory, calldata	Gibt den Speicherort an (dauerhaft, temporär, oder nur zur Eingabe).
Sichtbarkeit	public, private, internal, external	Legt die Zugänglichkeit von Funktionen und Variablen fest.
Funktionen	function name() public {}	Definiert eine Funktion mit einer bestimmten Sichtbarkeit.
	returns (Type)	Gibt den Rückgabewert einer Funktion an.
	view, pure	Spezifiziert Funktionen, die nur lesen (view) oder keine Zustandsänderungen durchführen (pure).
	payable	Markiert Funktionen, die Ether empfangen können.
Konstruktor	constructor() public {}	Initialisiert den Smart Contract beim Deployment.
Modifier	modifier onlyOwner {}	Definiert Wiederverwendbare Bedingungen für Funktionen.
	require(condition, message)	Prüft Bedingungen und wirft Fehler, wenn diese nicht erfüllt sind.
	assert(condition)	Überprüft Zustände und löst Fehler aus (für kritische Tests).
Zugriffskontrolle	onlyOwner	Ein häufig verwendeter Modifier für Besitzer-Privilegien.
Ereignisse	event EventName(Type value);	Definiert ein Event zur Protokollierung von Daten auf der Blockchain.
	emit EventName(value);	Löst ein definiertes Event aus.
Ether senden/empfangen	msg.value	Gibt die gesendete Ether-Menge in Wei an.
	address(this).balance	Gibt das Ether-Guthaben des Contracts zurück.
	payable(address).transfer(x)	Sendet Ether an eine Adresse.
	payable(address).send(x)	Alternative Methode zum Senden von Ether (liefert bool).
Schleifen	for, while, do-while	Kontrollstrukturen für Iterationen.
Bedingungen	if, else, require	Kontrollstrukturen für Bedingungen.
Vererbung	is	Ermöglicht Vererbung von Contracts.
	super	Ruft eine Funktion der Elternklasse auf.
Bibliotheken	library LibraryName {}	Definiert eine wiederverwendbare Bibliothek.
Fallback/Receive	fallback()	Funktion, die aufgerufen wird, wenn keine andere Funktion passt.
	receive()	Funktion, die explizit Ether empfängt.
Arithmetik	+, -, *, /, %	Mathematische Operatoren.
	SafeMath	Bibliothek für sichere Berechnungen.
Typecasting	uint256(x), address(y)	Konvertiert Typen explizit.