Chainlink Fundamentals

Solidity Programming

Solidity is a programming language used to write **smart contracts** that run on the **Ethereum blockchain**. These smart contracts are programs that automatically execute rules, handle money, and interact with other contracts without needing a middleman.

---

1. What is Solidity?

- A **high-level language** similar to JavaScript, Python, and C++
- Designed to run on the **Ethereum Virtual Machine (EVM)**
- Used to build **dApps, tokens, DeFi protocols, NFTs**, etc.

Main features:

- Static typing (types are checked at compile time)
- Inheritance and reusable code
- Custom data types (structs, mappings, enums)
- Libraries for modular code
- Access to low-level EVM features

---

2. Basic Structure of a Solidity Contract

Every Solidity file usually contains:

1. **License identifier** (SPDX)
2. **Version pragma** (Solidity version)
3. **Contract declaration**

A contract is like a **blueprint** that defines:

- What data it stores
- What actions it can perform

---

3. Variables and Data Storage

State Variables

- Stored permanently on the blockchain
- Cost gas to update

Visibility types:

- public – anyone can read
- private – only inside the contract
- internal – contract + child contracts

Constant & Immutable

- **constant**: value known at compile time, cheapest gas
- **immutable**: value set once in constructor, cheaper than normal variables

Use them whenever possible to **save gas and improve safety**.

---

4. Data Types

Value Types

- Store actual values
- Copies are created when assigned
- Examples: uint, int, bool, address, bytes32

Reference Types

- Store a **pointer** to data
- Changes affect original data
- Examples: string, array, mapping, struct, bytes

---

5. Storage Locations

Where data lives matters a lot in Solidity:

- **storage** → permanent, expensive

- **memory** → temporary, cheaper
- **calldata** → read-only, cheapest (best for function inputs)

Using the right location helps **optimize gas usage**.

---

6. Functions

Functions define what your contract can do.

Types of functions:

- view → reads data
- pure → no state access
- payable → can receive ETH
- constructor → runs once at deployment

Function visibility:

- public, private, internal, external

---

7. Transaction & Block Variables

Solidity gives access to blockchain data:

- msg.sender → who called the function
- msg.value → ETH sent
- block.timestamp → current time
- block.number → block count

Used for:

- Authentication
- Payments
- Time locks
- Block-based logic

---

8. Control Flow

- **if / else** → decision making
- **loops** → repeat logic (use carefully due to gas costs)

---

9. Error Handling

- require() → checks conditions
- **Custom errors** → cheaper and preferred for gas efficiency

If a condition fails, the transaction is **reverted**.

---

10. Events

- Used to **log important actions**
- Help frontends track contract activity
- Example: token transfers

Events do **not** store data permanently but are very useful.

---

11. Modifiers

- Reusable conditions for functions
- Commonly used for **access control** (e.g., only owner)

They help keep code **clean and secure**.

---

12. Interfaces

- Define **what functions must exist**, not how they work
- Used to interact with other contracts safely
- Essential for standardized systems like ERC20, ERC721

---

13. Best Practices

- Keep contracts simple
- Validate inputs before changing state
- Use clear names and comments
- Follow naming conventions
- Always consider gas costs

---

14. What is ABI?

ABI (Application Binary Interface) is how external apps talk to your smart contract.

- Describes available functions and data types
- Generated automatically when a contract is compiled
- Used by frontend apps (like React + ethers.js)

Think of ABI as the **API documentation for smart contracts**.

---

15. Next Steps

- Practice small contracts
- Use **Remix IDE**
- Read real-world contracts
- Study OpenZeppelin
- Learn gas optimization
- Build slowly and consistently

Smart Contract Libraries & Inheritance

After learning Solidity basics, **libraries** and **inheritance** help you write **reusable, cleaner, and cheaper** smart contracts. They prevent code duplication and improve security.

---

1. Libraries

Libraries are reusable utility code (like toolboxes) that many contracts can use.

What Libraries Are Best For

- Math operations
- Helper functions
- Shared logic used across contracts

Key Points About Libraries

- Written once, used everywhere
- Cannot store state variables
- Improve gas efficiency
- Reduce code duplication

How Libraries Are Used

- Call functions directly (Library.function())
- Attach functions to data types using using for

Types of Libraries

- **Embedded libraries** (internal functions): copied into contract
- **Linked libraries** (external functions): deployed once, reused by many contracts

---

2. Inheritance

Inheritance lets one contract reuse and extend another contract's code.

Why Use Inheritance

- Build on existing contracts
- Add or customize features
- Avoid rewriting common logic

A child contract:

- Automatically gets all public and internal functions
- Can add new functions
- Can modify existing ones

---

3. Function Overriding

Used when you want to **change behavior** from a parent contract.

Rules

- Parent function → marked virtual
- Child function → marked override
- super keyword lets you call the parent's version of a function

This allows you to **extend behavior instead of replacing it fully**.

---

4. Multiple Inheritance

Solidity allows inheriting from **multiple contracts**, but it must be handled carefully.

Important Things to Remember

- If multiple parents have the same function name, you must explicitly override it
- You must list all parent contracts in override(...)
- **Order matters**: the first parent listed is checked first when using super

Incorrect ordering can lead to unexpected behavior.

---

5. Real-World Usage (OpenZeppelin)

Most real projects use **OpenZeppelin**, a trusted library of secure smart contracts.

Common uses:

- Creating ERC20 / ERC721 tokens
- Adding mint, burn, fee, or access control features
- Extending standard contracts safely

Inheritance makes it easy to customize standard tokens without breaking security.

---

6. Adding Custom Logic with Inheritance

You can:

- Add transfer fees
- Add burn or mint functions
- Change token behavior by overriding functions

This keeps your code **modular and clean**.

---

7. Tools & Imports

- **OpenZeppelin Contract Wizard** helps generate production-ready contracts
- Solidity supports importing:
    - npm packages
    - specific contracts
    - local files

---

8. Key Takeaways

- **Libraries** → best for reusable utility logic
- **Inheritance** → best for extending contracts
- Use virtual, override, and super correctly
- Parent order matters in multiple inheritance

- OpenZeppelin contracts are highly recommended

---

9. Best Practices

- Keep inheritance shallow and simple
- Clearly document overridden functions
- Avoid unnecessary complexity
- Reuse well-audited, trusted code

Testnet Funds

To use **Chainlink services** in this course, you need to deploy and interact with smart contracts on **testnets** like **Sepolia** and sometimes **Base Sepolia**. For this, you need **testnet tokens**, not real money.

---

1. Why Testnet Funds Are Needed

- **Testnet ETH** → Used to pay gas fees
- **Testnet LINK** → Used to pay for Chainlink services (oracles, VRF, etc.)
- These tokens have **no real value** and are used only for testing

---

2. Getting Testnet LINK Tokens

Using the Chainlink Faucet

- Visit the **Chainlink Faucet**
- Connect your wallet
- Select the testnet (e.g., Sepolia)
- Request LINK tokens
- Confirm the transaction in MetaMask
- LINK will appear in your wallet after confirmation

---

3. Adding LINK to MetaMask

- Go to **Chainlink documentation**
- Find the LINK token address for your testnet
- Click **Add to wallet**
- Confirm in MetaMask

⚠️ You must add LINK separately for **each network** you use.

---

4. Adding Other Tokens to MetaMask (e.g., USDC)

If a token doesn't have an "Add to wallet" button:

1. Find the token's **contract address**
2. Switch MetaMask to the correct network
3. Go to **Tokens → Import tokens**
4. Paste the token address
5. Confirm the token details
6. Import the token

Now you can view and send that token using MetaMask.

---

5. Sending Tokens to Another Address

To send LINK or any other token:

1. Open MetaMask
2. Go to the **Tokens** tab
3. Select the token
4. Click **Send**
5. Paste the recipient's address
6. Enter the amount

7. Confirm the transaction

This is commonly used to **fund smart contracts** with LINK.

Key Takeaways

- You'll mainly use **Sepolia testnet**
- You need:
    - Testnet ETH for gas
    - Testnet LINK for Chainlink services
- Use faucets to get free test tokens
- Tokens must be added manually to MetaMask per network
- You can send tokens just like real crypto

Remix

What is Remix?

Remix is a **browser-based IDE** used to write, compile, deploy, and test **Solidity smart contracts**.
You don't need to install anything—just open it in your browser and start coding.

Why Remix is Useful

- Beginner-friendly and fast to use
- Built-in tools for smart contract development
- Works directly with MetaMask
- Great for learning and testing before using real blockchains

---

Main Features of Remix

- **File Explorer** → Create and manage files and folders
- **Editor** → Write Solidity code
- **Solidity Compiler** → Compile contracts and find errors
- **Deploy & Run Transactions** → Deploy and interact with contracts
- **Debugger** → Debug failed transactions
- **Plugins** → Add extra tools when needed

Remix Interface Overview

- **Left panel**: File Explorer
- **Center**: Code editor
- **Bottom**: Terminal (logs & errors)
- **Right sidebar**: Compiler and Deployment tools

Creating a Project in Remix

1. Create a **workspace** (like a project folder)
2. Create a **contracts** folder
3. Create a Solidity file (e.g., MyERC20.sol)
4. Start writing your smart contract code

Compiling Smart Contracts

- Open the **Solidity Compiler**
- Select the correct Solidity version
- Click **Compile**
- Fix any errors shown in the terminal
- Successful compile shows a green checkmark
- Compilation also generates the **ABI**, which is used by frontends and verification tools

---

Connecting Remix to MetaMask

To deploy on a real network:

1. Open MetaMask and select a network (e.g., Sepolia)
2. In Remix, choose **Injected Provider – MetaMask**
3. Approve the connection in MetaMask

Deploying Smart Contracts

1. Go to **Deploy & Run Transactions**
2. Select environment:
    - **Remix VM** → Local testing
    - **Injected Provider – MetaMask** → Real testnet/mainnet
3. Select the contract
4. Click **Deploy**
5. Confirm the transaction in MetaMask

After Deployment

- **Pin contracts** to keep them after refresh
- Copy the **contract address**
- Use **At Address** to reconnect to deployed contracts

Interacting with Smart Contracts

- Expand the deployed contract
- Call functions directly from Remix
- Use **transact** for state-changing functions
- Confirm transactions in MetaMask

- View outputs and logs in the terminal

Final Takeaway

Remix is a **powerful learning tool** that lets you:

- Write Solidity contracts
- Compile and debug them
- Deploy to testnets or mainnet
- Interact with contracts easily

Writing an ERC-20 Token

What is an ERC-20 Token?

An **ERC-20 (Ethereum Request for Comment)** is a standard for **fungible tokens** on Ethereum.
It defines common rules like:

- Transferring tokens
- Checking balances
- Allowing others to spend tokens on your behalf

Because of this standard, ERC-20 tokens work smoothly with wallets, exchanges, and DeFi apps.
Examples: **USDT, USDC, DAI**

Creating a Basic ERC-20 Token

You create your token using **OpenZeppelin**, a trusted library that provides secure, pre-built contracts.

Basic Token Features

- Token name: *My Cyfrin CLF Token*
- Symbol: *CLF*
- Inherits OpenZeppelin's ERC20 contract
- Has a mint function to create new tokens

What the Code Does

- ERC20(...) → Sets token name and symbol
- _mint(to, amount) → Creates new tokens and sends them to an address
- Anyone can mint tokens **at first** (not secure yet)

Why Access Control is Needed

Without restrictions:
❌ Anyone can mint unlimited tokens
❌ Token supply can be abused

So, we add **permissions (roles)** to control who can mint.

Adding Access Control with OpenZeppelin

What is AccessControl?

AccessControl lets you:

- Create **roles**
- Assign roles to addresses
- Restrict functions to specific roles

Creating a Minter Role

- A role is identified using a bytes32 value
- The role name is hashed for safety

bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

Restricting the Mint Function

Using the onlyRole modifier:

- Only addresses with MINTER_ROLE can mint tokens
- Unauthorized users are blocked automatically

Admin Role (DEFAULT_ADMIN_ROLE)

- OpenZeppelin provides a built-in admin role
- Admins can grant or revoke roles
- The **deployer** gets this role automatically in the constructor

---

Final Secure Token Behavior

✓ Only approved addresses can mint tokens
✓ Deployer is both **Admin** and **Minter**
✓ New minters can be added later using grantRole
✓ Token follows ERC-20 standards

What You'll Do Next

- Compile the contract in Remix
- Deploy it to a testnet
- Add the token to MetaMask
- View balances and transfer tokens using the wallet UI

Deploying Your ERC-20 Token

Now that your **ERC-20 token with permissions** is ready, the next step is to **deploy it on a testnet** and **add it to MetaMask** so you can view balances and transfer tokens.

Step 1: Compile the Contract

Before deployment, the contract must be compiled.

- Open MyERC20.sol in Remix
- Go to the **Solidity Compiler** tab
- Select compiler version **0.8.19 or higher**
- Click **Compile MyERC20.sol** (or press Ctrl + S / Cmd + S)
- A **green checkmark** means compilation was successful

Step 2: Deploy the Contract

To deploy on a real testnet (like Sepolia):

- Open **Deploy & Run Transactions**
- Select **Injected Provider – MetaMask**
- Connect MetaMask when prompted
- Choose MyERC20.sol from the contract dropdown
- Click **Deploy**
- Confirm the transaction in MetaMask

Deployment is Successful When:

✓ Remix terminal shows a green tick
✓ MetaMask shows a successful deployment transaction
✓ The contract appears under **Deployed Contracts** in Remix

📌 **Important**:

- Pin the contract in Remix so it stays available
- Copy and save the contract address for later use

Step 3: Check Token Supply & Balances

After deployment:

- Use totalSupply() → shows total tokens (initially 0)
- Use balanceOf(address) → shows token balance (also 0 initially)
- No tokens exist yet because minting hasn't been done

Step 4: Add Your Token to MetaMask

To view and send your token in MetaMask:

1. Copy the token contract address from Remix
2. Open MetaMask → **Tokens tab**
3. Click **Import tokens**
4. Paste the contract address
5. MetaMask auto-fills token details
6. Click **Import**

Now your token appears in MetaMask 🎉

What You Learned

✓ How to compile and deploy an ERC-20 contract
✓ How to connect Remix with MetaMask
✓ How to pin and reuse deployed contracts
✓ How to check token supply and balances
✓ How to add a custom token to MetaMask

Interacting with Your Token

Minting Tokens

Right now, your token supply is **0** because no tokens have been created yet. To create tokens, you must call the **mint** function.

Steps to mint tokens in Remix:

- Go to **Deploy & Run Transactions**
- Expand your deployed token contract
- Open the **mint** function
- Enter:
    - **to** → the wallet address that will receive the tokens
    - **amount** → token amount in wei
        - Example: To mint **100 tokens**, use 100000000000000000000 (because ERC-20 uses 18 decimals)
- Click **transact**
- Confirm the transaction in MetaMask

After minting:

- Call **balanceOf(address)** to check the wallet balance
- Or check your balance directly in **MetaMask → Tokens tab**

Token Approvals & Allowance

What is Token Approval?

Token approval allows **another address or smart contract** to spend your tokens on your behalf.
This is commonly used in **DeFi apps** (DEXs, staking, lending, etc.).

Approving Token Spending

- Use the **approve(spender, amount)** function

- This grants permission to a specific address to spend a certain number of your tokens

---

Checking Allowance

- Use **allowance(owner, spender)**
- This shows how many tokens the spender is still allowed to use

---

Oracle Concepts

Why Blockchains Need Oracles

Blockchains are **secure and deterministic**, meaning:

- The same input always gives the same output
- All nodes must agree on every transaction

Because of this design, blockchains **cannot directly access real-world data** like prices, weather, or sensor readings.
This limitation is called the **Oracle Problem**.

Without oracles:
❌ Smart contracts can't react to real-world events
❌ Blockchain apps are limited in functionality

What is a Blockchain Oracle?

A **blockchain oracle** is a system that **connects blockchains with off-chain (real-world) data**.

Oracles allow smart contracts to:

- Use live data (prices, weather, scores)
- Trigger actions based on real-world events
- Perform off-chain computation safely

What Oracles Do (In Simple Terms)

Oracles handle the full data flow:

1. Listen for a data request from a smart contract
2. Fetch data from external sources (APIs, sensors, websites)
3. Format the data so the blockchain can understand it
4. Verify and secure the data
5. Perform off-chain computation if needed
6. Send (broadcast) the data back to the blockchain
7. Deliver results to external systems if required

Types of Blockchain Oracles

- **Inbound Oracles** → Bring data *into* the blockchain (prices, weather)
- **Outbound Oracles** → Send data *from* blockchain to external systems
- **Consensus Oracles** → Combine data from many sources for accuracy
- **Cross-Chain Oracles** → Enable communication between different blockchains

---

Centralized vs Decentralized Oracles

Centralized Oracles

- Single data provider
- ❌ Single point of failure
- ❌ Easy to attack or manipulate

Decentralized Oracles

- Multiple independent nodes
- ✓ More secure and reliable
- ✓ Trust is distributed instead of centralized

When Oracles Are Used

Oracles are essential when smart contracts need:

- **External data** → Market prices, insurance data, sports results
- **Automation** → Trigger actions based on conditions or time
- **Cross-chain communication** → Interact with other blockchains
- **Verifiable randomness** → Fair gaming, NFTs, random selection

---

What is Chainlink?

Chainlink is a **decentralized oracle network (DON)** that connects smart contracts to **real-world data, off-chain computation, and other blockchains** in a secure and reliable way.

It solves the **Oracle Problem** by using **many independent oracle nodes** instead of a single data source. These nodes collect data, agree on the correct value, and send one verified result to the blockchain.

Why Chainlink is Trusted

1. Trust Problem

Issue: Blockchains can't trust a single external data source.
Solution: Chainlink uses **multiple independent nodes** that reach consensus.
➡️ No single party controls the data.

---

2. Accuracy Problem

Issue: How do we know the data is correct?
Solution: Chainlink:

- Uses reputable data sources
- Tracks node reputation

- Uses cryptographic signatures

---

3. Reliability Problem

Issue: What if a data source fails?
Solution: Since many nodes and sources are used, the system continues working even if some fail.

How Chainlink Works (Simple Flow)

1. A smart contract requests data or computation
2. Chainlink selects multiple trusted nodes
3. Nodes fetch data independently
4. Results are compared and aggregated
5. Final verified data is sent on-chain
6. Nodes are paid in **LINK tokens**

Major Chainlink Services

- **Data & Price Feeds**
  - Provides accurate on-chain prices (ETH/USD, BTC/USD, etc.)
  - Used heavily in **DeFi platforms**
  - Prevents price manipulation

- **Automation (Keepers)**
  - Automatically triggers smart contract functions
  - No human or centralized control needed
  - Used for liquidations, scheduled actions

- **CCIP (Cross-Chain Interoperability Protocol)**

- Enables secure communication between blockchains
- Allows token transfers and messages across chains

🔹 **Chainlink Functions**

- Runs **custom off-chain code**
- Sends verified results back to smart contracts
- Used for complex or expensive computations

🔹 **VRF (Verifiable Random Function)**

- Generates **provably fair randomness**
- Cannot be predicted or manipulated
- Used in NFTs, gaming, giveaways

🔹 **Data Streams**

- High-frequency, low-latency data
- Pull-based (data fetched only when needed)
- Ideal for fast-moving markets

🔹 **Proof of Reserve**

- Verifies that assets (like stablecoins) are fully backed
- Increases transparency and trust

Where Chainlink is Used

DeFi

- Accurate pricing for lending, borrowing, and trading
- Used by platforms like Aave and Uniswap

Gaming & NFTs

- Fair randomness for gameplay and NFT traits
- Prevents manipulation by developers or miners

The LINK Token – Why It Matters

LINK powers the Chainlink network:

- **Payment** → Node operators are paid in LINK
- **Security** → Nodes stake LINK as collateral to stay honest

This creates strong incentives for reliable and accurate data delivery.

What are Chainlink Data Feeds?

Chainlink Data Feeds let smart contracts safely use **real-world data** such as:

- Crypto prices (ETH/USD, BTC/USD, etc.)
- Interest rates and volatility
- Reserve balances
- Layer-2 sequencer status (whether an L2 is running properly)

They are widely used in **DeFi, NFTs, gaming, and financial apps**.

Types of Chainlink Data Feeds

- **Price Feeds**
  Provide accurate, aggregated crypto and asset prices.
- **SmartData Feeds**
  On-chain data designed for tokenized real-world assets (RWAs).
- **Rate & Volatility Feeds**
  Supply interest rates, curves, and volatility data.

- **L2 Sequencer Uptime Feeds**
  Tell you if an L2 network's sequencer is up or down (important for safety).

Main Components of Data Feeds

1. Consumer Contract

- This is **your smart contract**
- Uses AggregatorV3Interface to read data from Chainlink feeds
- Fetches data like prices directly on-chain

---

2. Proxy Contract

- Acts as a **pointer** to the correct aggregator
- Allows Chainlink to upgrade data sources without breaking your contract
- Your contract always talks to the proxy, not the aggregator directly

---

3. Aggregator Contract

- Managed by Chainlink
- Receives data from multiple oracle nodes
- Aggregates and stores verified data on-chain
- Makes data transparent and publicly verifiable

Why Chainlink Data Feeds are Important

- **Decentralized** → No single data source
- **Reliable** → Multiple nodes and sources
- **Upgradeable** → Proxy pattern avoids breaking changes
- **On-chain accessible** → Data usable in the same transaction

---

Simple Flow

1. Oracle nodes fetch data from many sources
2. Aggregator contract combines the data
3. Proxy contract points to the aggregator
4. Your smart contract (Consumer) reads the data

What are Chainlink Price Feeds?

Chainlink Price Feeds are decentralized data feeds that provide **secure and tamper-proof asset prices** (like crypto, commodities, and financial assets) to smart contracts.

They allow smart contracts to use **real-time market prices**, which is especially important in **DeFi**, where correct pricing is critical for trust and safety.

Why Price Feeds Matter

- Ensure **accurate pricing**
- Prevent **price manipulation**
- Enable **trust-minimized financial apps**
- Replace unreliable single-source price data

Common Use Cases

DeFi Protocols

- Used by platforms like **Aave** and **Compound**
- Helps decide lending, borrowing, liquidation, and trading prices
- Ensures loans stay properly collateralized

Stablecoins

- Maintain the stablecoin's **price peg**
- Provide correct valuation of backing assets

Derivatives & Prediction Markets

- Used to settle derivatives contracts
- Provide live market data for predictions

Finding Available Price Feeds

- Prices are shown as **asset pairs** (e.g., ETH/USD, EUR/USD)
- Available feeds depend on the **blockchain network**
- You can find all supported feeds at **data.chain.link**

⚠️ **Important:**
To use a Price Feed in a smart contract, you must know its **on-chain contract address**.

What is Chainlink Automation?

Chainlink Automation is a decentralized service that **automatically runs smart contract functions** when certain conditions are met or at scheduled times—without any manual action.

Why Automation is Needed

Smart contracts **cannot run on their own**. They need someone or something to trigger them.

Without automation:

- Conditions must be checked manually

- Functions must be triggered manually
- This is slow, unreliable, and inefficient

Chainlink Automation acts like a **24/7 operator** that:

- Constantly monitors your contract
- Automatically executes functions when needed
- Works reliably without human intervention

What is an "Upkeep"?

An **upkeep** is a registered task that Chainlink Automation watches and executes.

When you create an upkeep, you're saying:

"Monitor these conditions, and when they're met, call this function in my smart contract."

The conditions that cause execution are called **triggers**.

Types of Automation Triggers

1. Time-Based Triggers

- Run functions on a fixed schedule
- Example: every hour, daily, weekly

---

2. Custom Logic Triggers

- Use your own logic inside the smart contract
- Implement checkUpkeep to decide when execution should happen

---

3. Log (Event-Based) Triggers

- Trigger execution when a specific blockchain event occurs
- Useful for event-driven smart contracts

How Chainlink Automation Works (Architecture)

- A network of **Automation nodes** monitors registered upkeeps
- Nodes agree on when an upkeep should be executed
- They submit cryptographically signed reports
- A central **Automation Registry contract** verifies and executes the function

This system is powered by Chainlink's **OCR3 protocol**.

What is Time-Based Automation?

Time-based automation lets you **automatically call a smart contract function at fixed time intervals** (like every 5 minutes) using **Chainlink Automation**.

In this example, a smart contract function (count) is automatically executed every 5 minutes on **Ethereum Sepolia testnet**.

What You Need Before Starting

- Some **Sepolia ETH** (for gas fees)
- Some **LINK tokens on Sepolia** (to pay for automation)

What Are We Building?

- A **simple counter smart contract**
- It has a variable called counter
- Every time the count() function is called, the counter increases by 1
- Chainlink Automation will call this function automatically every 5 minutes

---

High-Level Steps

1. Create the Smart Contract

- Use **Remix IDE**
- Create a contract called TimeBased.sol
- Paste the provided counter contract code
- The count() function increments the counter

---

2. Deploy the Contract

- Compile the contract in Remix
- Connect MetaMask (Sepolia network)
- Deploy the contract
- Copy and save the deployed contract address

---

3. Verify the Contract on Etherscan

- Verification allows Chainlink Automation to read the contract's ABI
- Go to **Sepolia Etherscan**
- Verify and publish the contract source code
- After verification, Automation can call count()

---

4. Create a Time-Based Upkeep

- Go to the **Chainlink Automation app**
- Register a new upkeep
- Choose **Time-based trigger**
- Enter the contract address
- Select the count() function

---

5. Set the Time Schedule (Cron)

- Use this cron expression to run every 5 minutes:
- */5 * * * *
- Cron expressions define *when* a function runs

---

6. Configure Upkeep Details

- Give the upkeep a name
- Set gas limit (default is fine)
- Fund it with LINK (e.g., 5 LINK)
- Register the upkeep and approve transactions

What Happens After Setup?

- Chainlink Automation calls count() every 5 minutes
- The counter value increases automatically
- You can track:
    - Last execution time
    - LINK balance
    - Execution history
- You can see the updated counter value on **Etherscan**

Stopping or Managing Automation

- You can **pause** the upkeep anytime
- You can **withdraw remaining LINK**
- You can resume or delete it later

What is Custom Logic Automation?

Custom Logic Automation lets you **decide exactly when a smart contract function should run** by writing your own conditions inside the contract.

Instead of using a fixed schedule (like every 5 minutes), **your contract itself tells Chainlink when it's ready** to execute.

Key Idea Behind Custom Logic

- Your contract implements **two required functions**:
  - **checkUpkeep** → Checks conditions and returns true or false
  - **performUpkeep** → Runs the actual logic when checkUpkeep is true
- If checkUpkeep returns true, Chainlink Automation automatically calls performUpkeep

This makes automation **flexible and fully programmable**.

Example Used

- A counter contract named **CustomLogic**
- Counter increases **every 5 minutes**
- Time condition is written **inside the smart contract**, not in the Automation UI

How the Contract Works

Constructor

- Takes an _updateInterval (in seconds)
- Set to 300 seconds (5 minutes)
- Stores the deployment time using block.timestamp

---

checkUpkeep

- Checks whether 5 minutes have passed since the last update
- Returns:
    - true → upkeep should run
    - false → do nothing

---

performUpkeep

- Executes when checkUpkeep is true
- Increments the counter
- Updates the timestamp

Deploying the Contract

- Deploy on **Sepolia testnet**
- Pass 300 as the constructor argument
- Pin the deployed contract in Remix

Verifying the Contract

- Contract must be verified on **Etherscan**
- Since the contract imports other files:
    - **Flatten the contract**
    - Use the flattened code for verification

Registering Custom Logic Upkeep

- Go to **Chainlink Automation App**
- Select **Custom Logic trigger**
- Enter the deployed contract address
- Fund with LINK (e.g., 5 LINK)

- Confirm and register the upkeep

What Happens After Registration?

- Chainlink nodes regularly call checkUpkeep
- Once 5 minutes pass:
  - checkUpkeep returns true
  - performUpkeep runs automatically
- The counter value increases
- Execution history and LINK usage are visible in the Automation dashboard

---

Important Notes

- To manually test checkUpkeep or performUpkeep, pass **empty bytes** as input
- Automation keeps running until:
  - LINK balance runs out
  - The upkeep is paused

What is Log Trigger Automation?

Log Trigger Automation lets Chainlink automatically run a smart contract function **when a specific on-chain event is emitted**.

In short:

📢 **Event happens → Chainlink detects it → Function runs automatically**

This enables **event-driven automation**, which is very powerful in blockchain apps.

Example Setup (Two Contracts)

1️⃣ EventEmitter Contract

- Emits an event called **WantsToCount**
- The event is emitted when the function emitCountLog is called
- This event acts as the **trigger**

2️⃣ LogTrigger Contract

- Responds to the emitted event
- Keeps a **counter**
- Chainlink Automation calls its function when the event is detected

---

How Log Trigger Automation Works

1. A user calls emitCountLog in the **EventEmitter** contract
2. The contract emits the **WantsToCount** event
3. Chainlink Automation listens for this event
4. Automation calls the function in the **LogTrigger** contract
5. The counter increases automatically

Important Functions in LogTrigger

To work with log triggers, the contract must implement **ILogAutomation**, which requires:

checkLog

- Simulates whether the upkeep should run
- Returns data (performData) used during execution
- In this example, it passes the event sender's address

---

performUpkeep

- Runs when the event is detected
- Uses performData
- Increments the counter
- Emits a log showing who triggered the event

Deployment & Verification

- Deploy both contracts on **Sepolia**
- Pin them in Remix
- **Flatten the files**
- Verify both contracts on **Etherscan**

Creating the Log Trigger Upkeep

In the **Chainlink Automation App**:

- Select **Log trigger**
- Contract to automate → LogTrigger
- Contract emitting logs → EventEmitter
- Event → WantsToCount
- Fund with LINK (e.g., 5 LINK)
- Register the upkeep

Testing the Automation

- Call emitCountLog in the **EventEmitter** contract
- Chainlink detects the event
- Automation executes the upkeep
- The counter in LogTrigger increases by 1
- Execution appears in the upkeep history

What is Blockchain Interoperability?

Blockchain interoperability means different blockchains can **talk to each other and share data**.

This is done using **cross-chain messaging protocols**, which allow:

- One blockchain to **send data or tokens** to another
- Smart contracts on different chains to **work together**

This enables **cross-chain dApps**, which:

- Share state across blockchains
- Stay connected instead of running isolated copies on each chain

---

What is Token Bridging?

Token bridging allows users to move tokens from one blockchain to another.

General idea:

- Tokens are **locked or burned** on the source chain
- Equivalent tokens are **minted or unlocked** on the destination chain

This enables **cross-chain liquidity** and broader token usability.

Token Bridging Mechanisms

1️⃣ Lock and Mint

- **Source chain**: Tokens are locked in a smart contract
- **Destination chain**: Wrapped tokens are minted (1:1 value)
- On return: Wrapped tokens are burned → original tokens unlocked

📌 Wrapped tokens act like an **IOU**

2️⃣ Burn and Mint

- **Source chain**: Tokens are permanently burned

- **Destination chain**: New tokens are minted
- Same process happens when moving back

📌 Best for tokens with minting control (native tokens)

3️⃣ Lock and Unlock

- **Source chain**: Tokens are locked
- **Destination chain**: Tokens are released from a liquidity pool
- Requires liquidity on both chains

📌 Liquidity providers are usually rewarded

4️⃣ Burn and Unlock

- **Source chain**: Tokens are burned
- **Destination chain**: Tokens are unlocked from reserves
- Requires existing liquidity

📌 Combines burning with liquidity-based unlocking

What is Cross-Chain Messaging?

Cross-chain messaging allows blockchains to send **data and instructions**, not just tokens.

Used for:

- Syncing protocol state (governance, interest rates)
- Triggering smart contract functions on another chain
- Building advanced cross-chain applications

Security Considerations in Cross-Chain Systems

Cross-chain systems are more complex and introduce new risks.

They must balance:

- 🔐 **Security** – Protection from attacks
- 🤝 **Trust assumptions** – Reliance on validators or oracles (Chainlink minimizes this)
- ⚙️ **Flexibility** – Supporting different blockchain designs

📌 Cross-chain apps need **stronger security design** than single-chain apps.

Chainlink CCIP

Chainlink CCIP (Cross-Chain Interoperability Protocol) lets blockchains talk to each other securely. Using CCIP, developers can:

- Send **tokens** between blockchains
- Send **data/messages** between blockchains
- Send **tokens + instructions together** in one transaction

It is designed with very strong security because cross-chain actions are risky.

What Can CCIP Do?

1. Send Messages (Data)

- Send any kind of data to smart contracts on other blockchains
- Trigger actions like minting NFTs, rebalancing portfolios, or running logic
- Combine multiple actions in one cross-chain message

2. Transfer Tokens

- Move tokens safely across chains
- Send tokens to wallets or smart contracts

- Built-in limits to reduce risk
- Works smoothly with DeFi apps using a standard interface

3. Programmable Token Transfers

- Send tokens **with instructions**
- Useful for DeFi actions like lending, swapping, or staking
- Everything happens in a single cross-chain transaction

How Is CCIP Secure?

CCIP uses a **defense-in-depth security model**, meaning multiple security layers:

- Many **independent Chainlink nodes**
- **Three separate oracle networks (DONs)** to commit, verify, and execute transactions
- A separate **Risk Management Network (RMN)** that checks for threats
- Different software implementations to reduce bugs
- Can adapt to new or unexpected attacks

This makes CCIP one of the most secure cross-chain systems.

How CCIP Works (Architecture)

- **Sender**: A user or smart contract that starts the transaction
- **Router**: The main CCIP contract that:
    - Handles fees and approvals
    - Sends messages to the correct chain
    - Delivers tokens or data to the receiver
- **Receiver**: A smart contract or wallet on the destination chain
    - Only smart contracts can receive data

Fees in CCIP (Simple)

You pay **one fee on the source chain** (no need to pay again on the destination chain).

Total Fee = Blockchain Fee + Network Fee

Blockchain Fee

Covers gas needed on the destination chain:

- Gas to execute the transaction
- Extra cost for data (if sending messages)
- Token transfer costs
- Extra buffer for gas spikes (**Smart Execution**)

⚠️ If you set a high gas limit and don't use it fully, **unused gas is not refunded**.

Network Fee

- Pays Chainlink services (oracle nodes, risk checks)
- Depends on whether you send tokens, data, or both

CCIP Transaction Flow (Step-by-Step)

1. **Message Sent**
   A user or contract sends tokens, data, or both on the source chain.
2. **Source Chain Finality**
   CCIP waits until the transaction is final and cannot be reversed.
3. **Commit Phase**
   Oracle nodes batch transactions and store proof on the destination chain.
4. **Risk Check (RMN)**
   The Risk Management Network verifies and approves the batch.

5. **Execution**
   Another oracle network executes the transaction on the destination chain:
   - Tokens are minted/unlocked
   - Messages are delivered to the receiver
6. **Smart Execution**
   CCIP adjusts gas automatically to make sure execution succeeds (within ~8 hours).

Token Pool Contracts (Simple Meaning)

- Token Pools manage **token supply across chains**
- Each token on each chain has its own pool
- Depending on the bridge type, the pool:
  - **Locks & unlocks tokens**, or
  - **Burns & mints tokens**
- Token Pools are responsible for safe token handling

Why Finality Matters

- Finality means a transaction **cannot be reversed**
- Different blockchains have different finality times
- CCIP waits for proper finality before continuing
- This prevents fraud and double-spending

Transporter

Transporter is a user-friendly app that helps you send tokens (and messages) from one blockchain to another. It is built on **Chainlink CCIP**, which provides very high security for cross-chain transfers.

Transporter is developed with support from the **Chainlink Foundation and Chainlink Labs** and offers:

- A simple and intuitive interface
- 24/7 support
- A visual tracker to monitor your cross-chain transactions in real time

Because it uses CCIP, Transporter benefits from strong decentralized security and a separate risk management system.

What You Do in This Lesson

You use Transporter to **bridge USDC from Ethereum Sepolia to Base Sepolia**.

Behind the scenes:

- CCIP handles the cross-chain messaging
- For USDC, CCIP also uses **Circle's CCTP** to safely burn USDC on the source chain and mint it on the destination chain

---

Requirements Before Starting

You must have:

- **LINK tokens on Sepolia** (for fees)
- **USDC tokens on Sepolia** (test tokens from Circle faucet)
- LINK and USDC added to MetaMask on **both Sepolia and Base Sepolia**
- MetaMask wallet connected

How Bridging Works (High-Level Steps)

1. **Connect your wallet** to the Transporter testnet app.
2. **Accept the terms** and confirm the wallet connection.
3. Choose:

    - **From:** Ethereum Sepolia
    - **To:** Base Sepolia
    - **Token:** USDC
    - **Amount:** 1 USDC
4. **Approve USDC** (one-time approval for 1 USDC).
5. **Send the transaction** and confirm it in MetaMask.
6. USDC is:
    - **Burned on Sepolia**
    - Then **minted on Base Sepolia** after finality is reached.
7. Track the transaction using the **CCIP Explorer**.
8. After ~20 minutes, the status becomes **Success**.

Final Result

- Your **USDC balance on Base Sepolia increases by 1**
- The cross-chain transfer is completed securely using CCIP

Transferring Tokens Cross-chain in a Smart Contract – Part 1

What This Lesson Is About

This lesson teaches how to **send USDC from Sepolia to Base Sepolia using a smart contract** powered by **Chainlink CCIP**.
Instead of using a UI (like Transporter), you learn how to do the transfer **directly from Solidity code**.

What You Need Before Starting

- MetaMask with **Sepolia** and **Base Sepolia** networks added

- Test **ETH** on both networks
- Test **LINK on Sepolia** (to pay CCIP fees)
- Test **USDC on Sepolia** (from Circle faucet)
- LINK and USDC added to MetaMask

What You Build

You create and deploy a smart contract called **CCIPTokenSender** on **Sepolia** that:

- Takes USDC from a user
- Pays CCIP fees using LINK
- Sends USDC cross-chain to **Base Sepolia**

High-Level Workflow

1. Deploy CCIPTokenSender on Sepolia
2. Fund the contract with **LINK**
3. User approves the contract to spend their **USDC**
4. User calls transferTokens()
5. CCIP sends USDC to Base Sepolia securely

How the Smart Contract Works (Conceptually)

1. Contract Setup

- Uses **OpenZeppelin Ownable** for admin control
- Uses **SafeERC20** for safer token transfers.
    - **Code: *using SafeERC20 for IERC20;***
- Uses **Chainlink CCIP Router** for cross-chain messaging

---

2. Important Constants

The contract stores:

- CCIP **Router address**
- **LINK token** address (for fees)
- **USDC token** address (to transfer)
- **Destination chain selector** (Base Sepolia ID)

---

3. transferTokens() Function (Main Logic)

This function:

- Checks the user has enough **USDC**
- Builds a **CCIP message** describing:
    - Who receives the tokens
    - Which token (USDC)
    - How much to send
- Gets the **CCIP fee** in LINK
- Verifies the contract has enough LINK
- Transfers USDC from the user to the contract
- Approves CCIP Router to spend USDC and LINK
- Sends the cross-chain message using ccipSend()
- Returns a **messageId** for tracking

👉 Since the receiver is a wallet (EOA), **no gas limit is needed** on the destination chain.

---

4. Fees

- CCIP fees are paid in **LINK**
- Fees depend on:
    - Destination chain

- Gas costs
- Network conditions

---

5. Withdrawal Function

- The contract owner can withdraw any leftover USDC from the contract
- Prevents tokens from getting stuck

Important Things to Remember

- Users **must approve USDC first** before calling transferTokens
- The contract **must be funded with LINK**
- Each destination chain has a **different chain selector**
- Fees are dynamic and paid via CCIP

Deploying the Contract

1. Open **Remix**
2. Compile CCIPTokenSender.sol
3. Connect MetaMask (Sepolia)
4. Deploy the contract
5. Confirm the transaction in MetaMask

Once deployed, the contract is ready to send USDC cross-chain.

Transferring Tokens Cross-chain in a Smart Contract – Part 2

In this part, you **actually send USDC cross-chain** using the smart contract you built earlier.

You move **USDC from Sepolia → Base Sepolia** using **Chainlink CCIP**, step by step.

Step 1: Fund the Contract with LINK

- CCIP requires **LINK tokens** to pay cross-chain fees
- You send **~3 LINK** from your wallet to the deployed CCIPTokenSender contract
- This LINK is used to cover CCIP execution costs

👉 You verify the LINK balance using **Etherscan** to confirm the contract is funded.

Step 2: Approve the Contract to Spend USDC

- Your smart contract uses safeTransferFrom to take USDC from your wallet
- Before that, you must **approve** the contract to spend your USDC

Steps:

- Go to **USDC contract on Etherscan (Sepolia)**
- Call the approve function
- Set:
    - spender → CCIPTokenSender address
    - value → 1 USDC (1000000, since USDC has 6 decimals)
- Confirm the transaction

You then verify the approval using the allowance function.

Step 3: Execute the Cross-chain Transfer

Now the real transfer happens 🚀

- Go back to **Remix**

- Call transferTokens with:
    - _receiver: your wallet address
    - _amount: 1000000 (1 USDC)
- Confirm the transaction in MetaMask

At this point:

- USDC is **burned on Sepolia**
- CCIP message is sent
- Tokens will be **minted on Base Sepolia** after finality

Step 4: Track the Transfer Using CCIP Explorer

- Copy the **Sepolia transaction hash**
- Paste it into the **CCIP Explorer**
- You can see:
    - Source chain transaction
    - Destination chain transaction
    - Message status

⏳ Ethereum Sepolia takes ~20 minutes for finality.

When status changes to **Success** → tokens are delivered 🎉

Step 5: Check Your USDC on Base Sepolia

- Switch MetaMask network to **Base Sepolia**
- Open the **Tokens** tab
- Your USDC balance should increase by **1 USDC**

✅ Cross-chain transfer complete!

CCIP v1.5 & Cross-Chain Token (CCT)

CCIP v1.5 by Chainlink introduces the **Cross-Chain Token (CCT) standard**, which makes it much easier for developers to create and manage tokens that work across multiple blockchains. Developers no longer need Chainlink to manually approve or integrate each token—they can do everything themselves.

Why CCT Was Introduced

1. **Liquidity is fragmented**
   Tokens are often stuck on one blockchain, making it hard to use them across different chains without losing liquidity or trust.
2. **Lack of developer control**
   Earlier cross-chain solutions depended on third parties and manual approvals. CCT fixes this by giving developers full control.

Key Benefits of the CCT Standard

- 🚀 **Self-service**: Enable a token for cross-chain use in minutes.
- 🛠️ **Full developer control**: Developers own and manage their tokens, pools, logic, and rate limits.
- 🔐 **High security**: Uses Chainlink's oracle networks, risk management systems, and transfer limits.
- 🔁 **Programmable transfers**: Tokens and messages can be sent together in one transaction.
- ✅ **Audited token pools**: Chainlink provides audited contracts to safely handle cross-chain logic.
- 💸 **Zero slippage**: The amount sent on one chain is exactly what's received on the other.
- 🔌 **Works with existing ERC-20 tokens**: No need to redesign tokens or use risky bridges.

How CCT Works (Main Roles)

- **Token Owner**: Owns and controls the token contract.
- **CCIP Admin**: Similar authority as the owner, specifically for CCIP-related actions.
- **Token Administrator**: Handles cross-chain configuration and links tokens to token pools.
- **Token Pools**: Smart contracts that manage how tokens move between chains (mint, burn, lock, unlock).
  👉 They **coordinate cross-chain behavior** and don't always hold tokens like traditional liquidity pools.

Supported Cross-Chain Mechanisms

CCT supports multiple safe token movement methods:

- Mint & Burn
- Mint & Unlock
- Burn & Unlock
- Lock & Unlock

Each token must have a **Token Pool** on every chain, regardless of the method used.

How to Use the CCT Standard

1. **Deploy your token** on every chain you want to support.
2. **Deploy token pools** (custom or Chainlink-audited).
3. **Configure pools** (rate limits, security settings).
4. **Register everything** in the TokenAdminRegistry to link tokens with pools.

In Short

CCT makes cross-chain tokens easy, secure, and fully developer-owned.
It removes friction, avoids liquidity issues, supports existing ERC-20 tokens, and enables true multi-chain applications with zero slippage and high security.

Questions

What type of infrastructure primarily enables distinct blockchain networks to exchange information and interact with one another?

Cross-chain messaging protocols

Chainlink Local

Chainlink Local is a developer tool that lets you **run and test Chainlink services on your own computer**. It helps you build and test smart contracts locally before deploying them to testnets or mainnet.

What Does Chainlink Local Do?

- Lets you **simulate Chainlink services locally**
- Works with popular tools like **Hardhat, Foundry, and Remix**
- Allows testing advanced features like **CCIP cross-chain messages** without spending real gas

---

Key Features (In Short)

- 🧪 **Local testing** of Chainlink services on local blockchain nodes
- 🔁 **Forked networks** to test against real deployed Chainlink contracts
- 🔌 **Easy integration** with Hardhat, Foundry, and Remix
- 🚀 **Testnet-ready code** — no changes needed after local testing

---

Two Ways to Test

1. Local Testing (No Forking)

- Uses **mock Chainlink contracts**
- Runs on local nodes like Hardhat or Remix VM
- Best for **early development and basic testing**

2. Local Testing (With Forking)

- Forks real blockchains for realistic testing
- Interacts with actual deployed Chainlink contracts
- Useful for **advanced CCIP and cross-chain testing**
- Available only in **Hardhat and Foundry** (not Remix)

Why Developers Use Chainlink Local

- Test **CCIP token transfers and messages** locally
- Simulate **cross-chain interactions**
- Catch bugs early in a safe environment
- Move confidently to testnets after local testing

Using Chainlink Local

In this lesson, instead of sending CCIP messages on a real testnet, **Chainlink Local** is used to **test cross-chain messaging locally in Remix**. This makes testing faster, cheaper, and instant.

We send a simple cross-chain message like **"Hey there!"** from one contract to another.

Contracts Used

1. MessageSender

- Deployed on the **source chain**
- Sends a CCIP message
- Uses LINK to pay fees
- Sends **only a string message** (no tokens)

2. MessageReceiver

- Deployed on the **destination chain**
- Inherits CCIPReceiver
- Implements _ccipReceive to handle incoming messages
- Stores:
    - Last message ID
    - Last received text

How the Message Flow Works

1. MessageSender creates a CCIP message
2. Message is routed through Chainlink CCIP
3. MessageReceiver automatically receives it
4. Message data is decoded and stored
5. You can read the message using a getter function

Why Gas Limit Is Needed

- The gas limit is used to execute _ccipReceive on the destination chain
- That's why a **non-zero gas limit** is required

CCIP Local Simulator

To simulate CCIP locally, we deploy a helper contract:

CCIPLocalSimulator

- Sets up mock CCIP infrastructure
- Deploys mock Router, LINK token, and other services
- Provides configuration values like:
    - LINK token address
    - Source router
    - Destination router
    - Chain selector

Deployment Steps (High Level)

1. Switch Remix environment to **Remix VM (Cancun)**
2. Deploy CCIPLocalSimulator
3. Get configuration details from it
4. Load the LINK token using **At Address**
5. Deploy:
    - MessageSender (using source router + LINK)
    - MessageReceiver (using destination router)

Sending a Cross-Chain Message

1. Call sendMessage from MessageSender
2. Provide:
    - Destination chain selector
    - Receiver contract address
    - Message text (e.g. “Hey there!”)
3. Manually set gas limit to **3,000,000**
4. Send the transaction

⏱️ The message is delivered **almost instantly** because everything is local.

Verifying the Message

- Call getLastReceivedMessageDetails on MessageReceiver
- You'll see:
    - Message ID
    - The text you sent

Using Chainlink CCIP to Interact with Smart Contracts Across Blockchains

Earlier, CCIP was used only to **transfer tokens across chains**. In this section, we go one step further by **sending tokens and instructions together**, so actions can be performed automatically on the destination chain.

Examples of actions:

- Staking tokens
- Depositing into a vault
- Buying NFTs or other assets

What Are Cross-Chain Messages?

Cross-chain messages are **arbitrary data sent as bytes** along with tokens. They can contain:

- Simple text (e.g. "Hey there")
- Encoded function calls
- Numbers or parameters (e.g. balances, amounts)

This allows smart contracts on one chain to **trigger logic on another chain**.

High-Level Architecture

The setup involves **three smart contracts**:

1. Sender (Source Chain)

- Encodes a function call
- Sends USDC + encoded data using CCIP

2. Receiver (Destination Chain)

- Receives tokens and message
- Decodes the data
- Executes the function call

3. Vault (Destination Chain)

- Receives USDC deposits
- Acts as a placeholder for real protocols (staking, DeFi, etc.)

Vault Contract (Destination Chain)

The Vault is a simple contract with two functions:

- deposit: stores received USDC
- withdraw: sends USDC back to the user

It represents **any application** you might interact with after receiving tokens cross-chain.

Steps:

- Create IVault.sol interface for ABI access
- Deploy Vault.sol on **Base Sepolia**
- Pin the contract for easy reuse

Sender Contract (Source Chain)

The Sender contract:

- Transfers USDC cross-chain
- Sends an encoded instruction to call deposit on the Vault

Key Points

Dynamic Receiver

- Receiver must be a **smart contract**, not an EOA
- EOAs can receive tokens but **cannot execute function calls**

Encoded Function Call

- The deposit function is encoded using:
    - Function selector
    - Parameters (depositor address and amount)

This encoded data tells the Receiver:

- Which contract to call (Vault)
- Which function to execute (deposit)
- What parameters to pass

Gas Limit

- A gas limit is specified for execution on the destination chain
- This gas is used when CCIP calls _ccipReceive on the Receiver contract

---

Deployment Flow

1. **Deploy Sender** on **Sepolia**
2. Pin the Sender contract
3. Switch MetaMask to **Base Sepolia**
4. Deploy and pin the **Vault**
5. Note down all deployed contract addresses

Questions

When creating a contract intended to receive messages via Chainlink CCIP, what is a mandatory step when inheriting from the `CCIPReceiver` abstract contract?

Implementing the `_ccipReceive` function to handle incoming message data.

Within a function designed to handle incoming Chainlink CCIP messages, how is the ABI-encoded message payload typically converted back into its original data type (e.g., a string)?

By using `abi.decode` with the expected data type(s).

When sending a cross-chain message that includes instructions for execution on the destination network, why is it often necessary to specify the `gasLimit`?
To pay for the execution of the `_ccipReceive` function on the destination network.

Introduction to Chainlink Functions

What is Chainlink Functions?

Chainlink Functions allows smart contracts to securely access **off-chain data and computation** using Chainlink's decentralized oracle networks (DONs).

It enables smart contracts to:

- Fetch data from **external APIs**
- Perform **custom JavaScript computation off-chain**
- Return verified results **on-chain**

In short:
👉 *Chainlink Functions bridges Web3 smart contracts with the Web2 world.*

Why Chainlink Functions is Needed

Smart contracts **cannot access external data** on their own due to blockchain determinism.

Without Chainlink Functions:

- No API access
- No off-chain computation
- Limited real-world use cases

Chainlink Functions removes this limitation in a **trust-minimized and decentralized** way.

Core Benefits

1. API Connectivity 🌐

- Connect smart contracts to **any public or private API**
- Aggregate, transform, and return data on-chain
- Enables real-world applications like weather insurance, sports-based NFTs, and real-time analytics

---

2. Custom Off-Chain Computation ⚙️

- Write **custom JavaScript code**
- Execute logic off-chain in a **serverless environment**
- Saves gas and avoids expensive on-chain computation
- Developers focus on logic, not infrastructure

---

3. Decentralized Infrastructure 🔐

- Powered by **Chainlink Decentralized Oracle Networks (DONs)**
- Multiple nodes execute the same code independently
- Eliminates single points of failure
- Maintains blockchain-level security guarantees

---

4. Off-Chain Consensus via OCR 📡

- Results from all nodes are aggregated using **Offchain Reporting (OCR).** The OCR protocol allows nodes to aggregate their observations into a single report offchain using a secure P2P network. A single node then submits a transaction with the aggregated report to the chain. Each report consists of many nodes' observations and has to be signed by a quorum of nodes. These signatures are verified onchain.
- Final output is consensus-based
- Prevents data manipulation by malicious nodes

How Chainlink Functions Works (Flow)

1. **Request Sent**
    - Smart contract sends JavaScript code + parameters
2. **Distributed Execution**
    - Each DON node runs the code independently
3. **Aggregation**
    - Results are combined using OCR
4. **Response Returned**
    - Final verified result is delivered on-chain

✓ Secure
✓ Trust-minimized
✓ Decentralized

Key Features

Decentralized Computation

- Off-chain execution with on-chain trust guarantees

Threshold Encryption for Secrets 🔑

- Securely include API keys or secrets
- Secrets are encrypted and decrypted only via **multi-party computation**
- No single node can access secrets alone

Subscription-Based Payments 💳

- Fund a **Chainlink Functions subscription** using LINK
- Pay only when requests are fulfilled
- Simple and predictable cost model

When to Use Chainlink Functions

Chainlink Functions is ideal for:

- **Public data access**
  - Weather, sports results, statistics
- **Data processing**
  - Sentiment analysis, calculations, aggregations
- **Authenticated APIs**
  - IoT devices, enterprise systems
- **Decentralized storage**
  - IPFS or off-chain databases
- **Web2 ↔ Web3 integration**
  - Hybrid dApps
- **Cloud service access**
  - AWS S3, Firebase, Google Cloud Storage

Important Considerations ⚠️

- **Self-service responsibility**
  - You must review APIs and JavaScript logic
- **No liability**
  - Chainlink and node operators are not responsible for faulty APIs or code
- **Data quality**
  - Accuracy depends on the source you choose
- **Licensing**
  - Ensure API usage complies with provider terms

Chainlink Functions Playground

What is Chainlink Functions Playground?

The **Chainlink Functions Playground** is an online sandbox tool that lets you **test JavaScript code for Chainlink Functions directly in your browser**.

You can:

- Run JavaScript off-chain
- Call external APIs
- See results instantly
  👉 **No smart contract deployment required**

It is the easiest way to learn and experiment with Chainlink Functions before going on-chain.

Why Use the Playground?

- Test your **JavaScript logic safely**

- Experiment without **gas fees or LINK**
- Debug API calls easily
- Verify outputs before integrating with smart contracts

Key Features

- **JavaScript Execution**: Run custom JS code off-chain
- **API Testing**: Call public APIs (HTTP requests)
- **Arguments Support**: Pass dynamic inputs to your code
- **Secrets Handling**: Securely test private data like API keys
- **Instant Results**: See output and logs in real time

Playground Input Sections

1. Source Code

- Write or paste the JavaScript code to execute

2. Arguments

- Dynamic values passed to your JS code
- Example: character ID, city name, token price

3. Secrets

- Private values (API keys, credentials)
- Injected securely at runtime

How to Run Code

1. Enter JavaScript in **Source Code**
2. Add values in **Arguments** (if needed)
3. Add **Secrets** (optional)
4. Click **Run Code**

5. View:
    - Output (returned result)
    - Console logs (debug info)

Example: Calling an External API

Star Wars API Example

- Uses a **character ID** as input
- Makes an HTTP request to a public API
- Returns character data (e.g., Luke Skywalker)

Steps:

1. Select **Star Wars characters** from example code
2. Enter 1 as argument
3. Click **Run Code**
4. Receive Luke Skywalker's data instantly

Why This Is Useful

- Confirms your API logic works
- Avoids smart contract redeployments
- Reduces debugging time
- Lowers learning barrier for Web3 + Web2 integration

Building a Chainlink Functions Consumer Smart Contract

In this lesson, we build a **smart contract** that uses **Chainlink Functions** to fetch data from an **external API** (weather data) and bring it **on-chain**.

The code is advanced, so it's okay if you don't understand everything. You can either copy-paste it to see how it works or study it step by step to learn integration.

Prerequisites

Before starting, you need:

- Sepolia testnet **ETH**
- Sepolia testnet **LINK**
- LINK token added to **MetaMask**

What Are We Building?

A **Chainlink Functions Consumer contract** that:

- Takes a **city name**
- Calls an external **weather API**
- Gets the **temperature**
- Stores and emits the result on-chain

Setup in Remix

1. Create a new workspace named **"Functions"**
2. Create a contracts folder
3. Create FunctionsConsumer.sol
4. Paste the provided contract code from the course repo

Key Parts of the Smart Contract

1. Imports

- FunctionsClient: lets the contract communicate with Chainlink Functions

- FunctionsRequest: helps create and manage requests

The contract **inherits FunctionsClient**, which requires the **Chainlink Router address** in the constructor.

---

2. State Variables

The contract stores:

- Last requested city
- City currently being requested
- Latest temperature
- Latest request ID
- API response and error data

This helps track requests and responses.

---

3. Constant Variables

These include:

- **Router address** (Sepolia-specific)
- **DON ID** (Chainlink network handling requests)
- **Gas limit** for the callback
- **JavaScript source code**

The JavaScript:

- Takes a city name as input
- Calls the wttr.in weather API
- Returns the temperature as a string

---

4. Events and Errors

- An **event** logs successful responses
- A **custom error** handles unexpected request IDs

---

getTemperature Function

This is the main function users call.

What it does:

1. Accepts a **city name** and **subscription ID**
2. Initializes a Chainlink Functions request using JavaScript code
3. Passes the city name as an argument
4. Sends the request to Chainlink
5. Stores and returns the request ID

This request is funded using a **Chainlink Functions subscription**.

---

fulfillRequest Function (Callback)

This function is called automatically by Chainlink after the API request finishes.

What it does:

1. Confirms the response matches the latest request
2. Stores the response and error data
3. Saves the temperature and city
4. Emits an event with the result

The **gas limit** defined earlier is used to control how much gas this function can consume.

Questions

When utilizing Chainlink Functions to connect smart contracts to external APIs or execute custom off-chain code, who typically bears the responsibility for ensuring the quality and reliability of external data sources and the correctness of submitted code?

The user or developer initiating the request.

Chainlink Functions Subscription

In this lesson, we create a **Chainlink Functions subscription**. This subscription is required to **fund, manage, and track** Chainlink Functions requests made by smart contracts.

Why Do We Need a Subscription?

A Chainlink Functions subscription:

- Pays for Chainlink Functions requests using LINK
- Manages which smart contracts can make requests
- Tracks request history and LINK usage

Creating a Chainlink Functions Subscription (Sepolia)

Steps:

1. Go to **functions.chain.link**
2. Connect your wallet (ensure you're on **Sepolia**)
3. Click **Create Subscription**
4. Enter your email and optional subscription name
5. Accept the **Terms of Service** (sign via MetaMask)
6. Approve the subscription creation
7. Sign again to link your email and confirm ownership

After these steps, your subscription is created.

Funding the Subscription

1. Click **Add Funds**
2. Enter **5 LINK** (testnet)
3. Confirm and sign the transaction in MetaMask

Once completed, your subscription is funded and ready to use.

Adding a Consumer Smart Contract

To allow your smart contract to use the subscription:

1. Click **Add Consumer** in the subscription dashboard
2. Copy your deployed FunctionsConsumer contract address from Remix
3. Paste it as the **Consumer address**
4. Sign the MetaMask transaction

Now, your smart contract is officially linked to the subscription.

Sending a Chainlink Functions Request

1. Go back to **Remix**
2. Open the deployed FunctionsConsumer contract
3. Call getTemperature with:
    - City: London
    - Subscription ID: your copied subscription ID
4. Click **Transact** and sign in MetaMask

Viewing the Results

- The request appears as **Pending** in the subscription dashboard
- Once completed, it moves to the **History** section
- In Remix, call:
    - s_lastCity
    - s_lastTemperature

You'll see the returned weather data on-chain.

Introduction to Chainlink VRF

Why Randomness Is Hard On-Chain

Blockchains are **deterministic** systems.
This means:

- Same input → same output
- Every node must reach the same result to maintain consensus

Because of this, **true randomness cannot be generated directly on-chain**.

Problems with On-Chain Randomness

If randomness is predictable or manipulable:

- Validators or miners can influence outcomes
- Smart contracts can be exploited
- Users can game lotteries or NFT mints
- Application fairness is compromised

Examples of bad randomness:

- block.timestamp
- blockhash
- msg.sender

What Is Chainlink VRF?

Chainlink VRF provides **provably fair, cryptographically secure randomness** for smart contracts.

It allows contracts to:

- Request random numbers
- Verify that the randomness was not tampered with
- Use the randomness safely on-chain

No party (oracle, miner, developer, or user) can manipulate the result.

How Chainlink VRF Works (Step-by-Step)

1. A smart contract requests randomness
2. Chainlink generates random value(s) called **random words**
3. A **cryptographic proof** is generated
4. The proof is published on-chain
5. The smart contract verifies the proof
6. Random values are safely consumed

✅ Result: **Provably random, tamper-proof randomness**

Key Use Cases of Chainlink VRF

- 🎮 **Blockchain Games** – fair loot drops and outcomes
- 🎨 **NFTs** – random trait generation and rarity
- 🏆 **Lotteries & Giveaways** – fair winner selection
- 🗳️ **DAO Governance** – unbiased committee selection
- 🔁 **Dynamic NFTs** – evolving traits based on randomness
- 📦 **Airdrops** – fair recipient sampling

Methods to Use Chainlink VRF

Chainlink VRF supports **two funding models**:

1️⃣ Subscription Method

A shared subscription is funded with **LINK or native tokens**, and multiple contracts can use it.

Advantages:

- One subscription → multiple consumer contracts

- Fees deducted **after fulfillment**
- Better gas efficiency
- Supports more random words per request
- Easier long-term cost management

Best For:

- Games
- NFT platforms
- DeFi protocols
- Apps with frequent randomness needs

2️⃣ Direct Funding Method

Each smart contract directly pays for randomness.

Advantages:

- No subscription setup
- Simple implementation
- Costs easily attributed per contract
- Can pass VRF cost to users

Best For:

- One-time NFT drops
- Giveaways
- Lotteries
- Infrequent randomness use

Security Considerations for Developers

- Never assume randomness arrives instantly
- Handle callback functions carefully

- Protect against re-entrancy in fulfillment
- Avoid depending on randomness in the same transaction
- Follow official VRF security guidelines

📘 Developers should review:
Chainlink VRF Security Best Practices

Creating a Chainlink VRF Subscription

To use **Chainlink VRF with the subscription method**, you first need to create and fund a VRF subscription. This subscription will pay for all your randomness requests.

Prerequisites

Before starting, make sure you have:

- **Sepolia ETH** (for gas fees)
- **Sepolia LINK** (to pay for VRF requests)

Both can be obtained from Chainlink faucets.

Steps to Create a VRF Subscription

1. Go to the **Chainlink VRF App** and connect your wallet.
2. Click **Create Subscription**.
3. (Optional) Add a subscription name and your email.
4. Confirm the transaction in MetaMask to create the subscription.
5. Sign a message to verify that you are the subscription owner.

Once confirmed, your **VRF subscription is created**.

Funding the Subscription

1. Click **Add funds**.
2. Enter **10 LINK** as the funding amount.
3. Confirm the transaction to transfer LINK to the subscription.

This LINK will be used to pay for random number requests.

Preparing to Use VRF

- Copy and save the **Subscription ID**.
- This ID will be required when your smart contract requests randomness.
- You can now **add consumer smart contracts** to this subscription.

Chainlink VRF in a Smart Contract

This lesson demonstrates how to use **Chainlink VRF** inside a smart contract to generate **secure and fair randomness**. The example project simulates **rolling a dice** and assigning a **Hogwarts house** based on the random result.

What This Smart Contract Does

- Requests a **verifiable random number** from Chainlink VRF
- Uses that number to simulate a **4-sided dice roll**
- Assigns the user to one of four Hogwarts houses:
    - Gryffindor
    - Hufflepuff
    - Slytherin
    - Ravenclaw

This type of logic is commonly used in **games, NFTs, lotteries, and giveaways**.

High-Level Flow (How It Works)

1. **User calls rollDice()**
    - The contract sends a randomness request to Chainlink VRF.
    - The request is linked to the user's address.
2. **Chainlink VRF generates randomness**
    - A random number is generated off-chain.
    - A cryptographic proof ensures it cannot be manipulated.
3. **Callback function is triggered**
    - The random number is returned to the contract.
    - The result is converted into a value from 0–3.
4. **House is assigned**
    - Each number maps to a Hogwarts house.
    - The result is stored on-chain.
5. **User checks their house**
    - Calling house(address) returns the assigned house name.

Why Chainlink VRF Is Used

- Blockchains cannot generate true randomness on their own
- Chainlink VRF provides:
    - **Unpredictable randomness**
    - **On-chain verification**
    - **Tamper-proof results**
- Ensures fairness in games and applications

Subscription Model Used

- The contract uses a **VRF subscription**
- LINK tokens from the subscription pay for randomness requests
- Multiple contracts can use the same subscription
- The consumer contract must be **added to the subscription**

Deployment & Usage Steps (Simplified)

1. Create and fund a **VRF subscription**
2. Deploy the **HousePicker** smart contract with the subscription ID
3. Add the contract as a **consumer** in the VRF app
4. Call rollDice() to request randomness
5. Call house() to see the assigned Hogwarts house

Key Takeaway

This lesson shows how to:

- Safely generate random numbers in smart contracts
- Use Chainlink VRF with the **subscription model**
- Build fair, secure, and trustless applications

Chainlink Data Streams

Chainlink Data Streams is a **high-speed, pull-based oracle system** that provides **real-time, low-latency market data** to blockchain applications. It is designed for **time-sensitive financial use cases** like trading and derivatives.

Core Components of Chainlink Data Streams

1. Chainlink Decentralized Oracle Network (DON)

- A group of independent Chainlink nodes
- Nodes collect price data from multiple data providers
- They agree on the **median price**
- The data is **signed** and sent to the Data Streams network
- Reports are delivered **very frequently** (often in less than a second)

---

2. Data Streams Aggregation Network

- Stores signed price reports
- Uses **active-active multi-site deployment**
    - Multiple locations run at the same time
    - Ensures high availability and fault tolerance
- Data can be accessed in two ways:
    - **Streams Trade** → Chainlink Automation fetches data for the dApp
    - **Streams Direct** → dApp fetches data directly via API

---

3. Chainlink Verifier Contract

- A smart contract that **verifies the DON's signature**
- Ensures the data:
    - Is authentic
    - Has not been altered
- Provides cryptographic security before the data is used on-chain

---

Available Data Streams

- Primarily provide **market price data**
- Prices are shown as **asset pairs** (example: ETH/USD)
- Availability depends on:

    - Blockchain network
    - Asset pair
- Supported pairs and chains can be viewed on **data.chain.link → Data Streams**

Key Use Cases

1. Perpetual Futures

- Enables fast and fair pricing for perpetual contracts
- Low latency reduces front-running risks
- Allows on-chain trading platforms to compete with centralized exchanges

---

2. Options Trading

- Provides precise price data for accurate settlement
- Helps manage risk dynamically using liquidity data
- Improves execution and reliability of options contracts

---

3. Prediction Markets

- Faster price updates allow quicker market reactions
- Improves accuracy and trust in outcome settlements
- Supports real-time event-based markets

Why Chainlink Data Streams Matter

- Delivers **fast, frequent, and accurate data**
- Maintains **security, transparency, and decentralization**
- Makes advanced financial products work efficiently on-chain

---

Chainlink Data Streams – Streams Trade

Streams Trade is an implementation of **Chainlink Data Streams** that uses **Chainlink Automation (Log Trigger)** to fetch **high-frequency off-chain price data** whenever a specific on-chain event occurs.

⚠️ **Important Note**
This feature requires **special access from the Chainlink team**, so most users cannot fully reproduce it. However, understanding the workflow is still very useful.

How Streams Trade Works (Step-by-Step)

1. A smart contract emits an **event** (for example, when a user performs an action).
2. **Chainlink Automation** detects this event using a **Log Trigger**.
3. Automation calls the checkLog function of another contract.
4. checkLog intentionally **reverts with a StreamsLookup error** (as per **EIP-3668**) to request off-chain data.
5. Chainlink nodes fetch a **signed report** from the Data Streams Aggregation Network.
6. Automation sends the data to the performUpkeep function.
7. The smart contract **verifies the data on-chain** using the Chainlink Verifier.
8. The verified price is stored on-chain.

➡️ Result: **Fast, secure, and verifiable price data** is available inside the smart contract.

Smart Contracts Used

1. LogEmitter Contract

- Simple contract
- Has one function: emitLog

- Emits an event when called
- This event **triggers Chainlink Automation**

📌 In real dApps, this could represent actions like trading, staking, or borrowing.

---

2. StreamsUpkeep Contract

Handles all Automation and Data Streams logic:

- Detects emitted logs
- Requests off-chain data
- Verifies signed price reports
- Stores the final price on-chain

Key Functions Explained Simply

checkLog

- Called by Automation when an event is emitted
- **Reverts with StreamsLookup**
- This tells Chainlink:
    - Which price feed to fetch (e.g., ETH/USD)
    - Which timestamp to use

➡️ This revert is intentional and follows **EIP-3668**.

---

checkCallback

- Runs **off-chain**
- Receives the fetched data
- Passes it to performUpkeep

---

performUpkeep

- Runs **on-chain**
- Main processing function
- What it does:
    - Decodes the data
    - Checks report version (V3 or V4)
    - Pays verification fees
    - Verifies report authenticity
    - Extracts price
    - Stores it in lastDecodedPrice

---

checkErrorHandler

- Optional error-handling function
- Used if something goes wrong during lookup
- In this example, it always retries fetching data

Report Versions

Chainlink Data Streams support two formats:

- **V3 Reports** → Crypto assets (e.g., ETH/USD)
- **V4 Reports** → Real-world assets (RWAs)

The contract automatically detects and decodes the correct format.

Deployment & Setup (High-Level)

1. Deploy both contracts on **Sepolia**
2. Fund StreamsUpkeep with **LINK**
3. Verify the contract on **Etherscan**

4. Register a **Log Trigger Upkeep** in Chainlink Automation
5. Fund the upkeep with LINK
6. Call emitLog to trigger the full flow

Viewing the Price

- The fetched ETH/USD price is stored in:
- lastDecodedPrice
- Uses **18 decimals**
- Example:
- 248412100000000000 → $2,484.121

Why Streams Trade Is Important

- Ultra-fast price updates (sub-second latency)
- Pull-based (data fetched only when needed)
- Secure and tamper-proof
- Ideal for:
    - Perpetual trading
    - Options
    - Prediction markets
    - High-frequency DeFi apps

One-Line Summary

Streams Trade uses Chainlink Automation to fetch and verify ultra-fast off-chain price data whenever an on-chain event occurs, ensuring speed, security, and decentralization.

Chainlink Proof of Reserve (PoR)

Chainlink Proof of Reserve (PoR) is a system that **automatically checks whether digital tokens are fully backed by real assets**. It connects **on-chain smart contracts** with **on-chain and off-chain reserve data** using Chainlink's **decentralized oracle network (DON)**.

It is mainly used for:

- Stablecoins
- Wrapped tokens
- Tokenized real-world assets (gold, bonds, etc.)

What is Collateralization?

Collateral is an asset locked as security to back another asset or loan.

Examples:

- ETH locked to borrow stablecoins
- USD in a bank backing a stablecoin
- Gold stored in a vault backing a token

If collateral is missing, the system becomes risky.

What is Proof of Reserve?

Proof of Reserve verifies that **every issued token is fully backed by reserves**.

Traditional PoR relies on **manual audits**, which are:

- Slow
- Centralized
- Not real-time

Chainlink PoR improves this by providing:

- Automated

- Real-time
- Decentralized reserve verification

Why Proof of Reserve is Important

Lack of transparency in crypto platforms has caused:

- Misuse of customer funds
- Hidden insolvency
- Market manipulation
- Platform collapses

Chainlink PoR prevents this by **continuously verifying reserves**, reducing trust issues and counterparty risk.

How Chainlink Proof of Reserve Works

1. Chainlink nodes **monitor reserve wallet addresses**
2. Nodes **verify off-chain reserves** using APIs and custodial data
3. Data is **aggregated and validated** by the oracle network
4. **Cryptographic proofs** of reserves are created
5. Verified data is **published on-chain**
6. Smart contracts can **automatically react**
   (e.g., stop minting if reserves fall)

Secure Mint Feature

Secure Mint ensures that:

- New tokens are minted **only when enough reserves exist**
- Prevents infinite minting and fraud
- Strengthens stablecoin and asset security

Technical Overview

- Uses **Chainlink PoR Data Feeds**
- Powered by decentralized oracle networks
- Updates based on thresholds or time intervals
- Works reliably even during market volatility
- Easily integrated into smart contracts

Key Benefits of Chainlink PoR

- Real-time transparency
- Reduced counterparty risk
- Decentralized and tamper-resistant
- Automated reserve verification
- Improved market stability
- Multi-chain support
- Scalable across asset types

Assets Covered by PoR Feeds

- **Fiat-backed stablecoins** (e.g., COP-backed stablecoin)
- **Treasury-backed stablecoins** (US T-Bills)
- **Fixed-income assets**
- **Tokenized commodities** (e.g., gold)

Use Cases

Stablecoins

Ensures every stablecoin is backed 1:1 by real assets, reducing de-pegging risk.

Wrapped Tokens

Verifies wrapped assets (like WBTC) are fully backed by original assets.

Tokenized Commodities

Proves that digital gold or silver tokens are backed by real vault holdings.

Cross-Chain Assets

Confirms backing of assets bridged across blockchains.

DeFi Protocols

Allows automatic risk control and collateral monitoring in lending and synthetic platforms.

One-Line Summary

Chainlink Proof of Reserve provides automated, real-time, decentralized verification that digital assets are fully backed by real reserves, improving transparency, security, and trust in the crypto ecosystem.
