

레드스톤(RedStone)

해당 번역본은 이용자가 디지털자산 관련 정보를 편리하게 확인할 수 있도록
단순 참고용으로 번역된 것이며, 투자 권유를 목적으로 하지 않습니다.
그 내용이 정확하지 않을 수 있으므로 정확한 정보 습득을 위해서는
원문을 참고하시거나 원문 작성 주체 측에 문의하시기 바랍니다.
두나무는 해당 번역본 내용의 진실성 및 정확성을 보장하지 않고,
제공된 정보에 의한 투자 결과에 대하여 어떠한 책임도 지지 않으며,
투자 손실은 투자자에게 귀속됩니다.

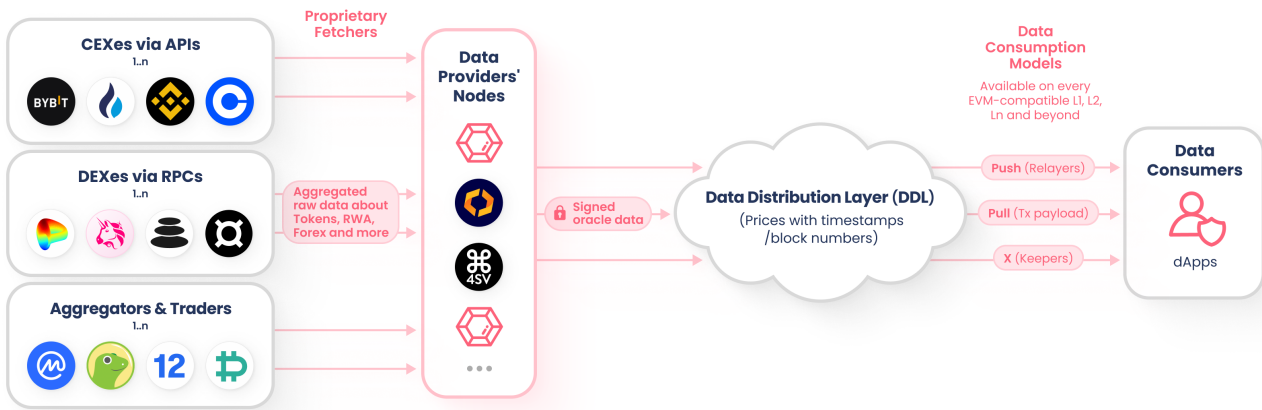
레드스톤(RedStone)이란 무엇인가?

소개

탈중앙화 애플리케이션(dApp)이 자주 업데이트되며, 신뢰할 수 있고, 보안이 강화된 데이터 피드에 접근해야 할 수요가 점점 커지고 있다. 레드스톤(RedStone)은 이러한 수요를 직접 충족하는 선도적인 블록체인 데이터 제공자이다. 100개 이상의 dApp이 신뢰하며 수십억 달러의 가치를 보호하고 있는 레드스톤은 차세대 dApp 빌더를 지원하기 위해 맞춤형이면서도 비용 효율적인 데이터 스트림을 제공한다. 또한 레드스톤은 전체 블록체인 생태계 전반에서 블록체인과 레이어 2 확장 솔루션에 데이터 피드를 제공하며, 이는 EVM 호환 환경과 비EVM 환경 모두에서 사용할 수 있다. 기존 블록체인 오라클 시스템의 모델은 여러 가지 비효율성을 겪고 있으며, 레드스톤은 이를 근본적으로 해결하도록 설계되었다. 따라서 레드스톤은 기존과 차별화된 독창적인 오라클 서비스이다.

멀티체인 데이터 전달

레드스톤의 모듈형 아키텍처는 데이터 수집과 데이터 전달을 분리하여 블록체인 네트워크 전반에 걸쳐 효율적으로 확장 가능한 유연한 시스템을 만든다. 이 설계 덕분에 동일한 가격 피드를 핵심 인프라 수정 없이 모든 지원 체인에 배포할 수 있다. 각 통합마다 새로운 노드를 배포할 필요가 없으므로 레드스톤은 더 빠르고 비용 효율적인 확장을 달성하면서도 모든 네트워크에서 일관된 보안 표준을 유지한다. 이 아키텍처는 이미 검증된 접근 방식으로, 레드스톤이 70개 이상의 블록체인 네트워크로 확장하는 데 기여했다.



독창적인 데이터 피드

레드스톤의 유연한 데이터 프레임워크는 표준 암호화폐 가격 피드를 넘어, 새롭게 등장하는 금융 상품과 특수화된 디지털 자산을 지원한다. 모듈형 아키텍처 덕분에 유동 리스테이킹 토큰(LRT), 비트코인 디파이 파생상품(BTCFi), 다양한 실물자산(RWA)과 같은 복잡한 금융 상품에 대해 맞춤형 검증 매개변수와 집계 방식을 바탕으로 신뢰할 수 있는 데이터를 제공할 수 있다. 레드스톤은 기관급 데이터 제공자 연결과 전통적인 암호화폐 시장 데이터를 결합하여 이러한 특수 피드를 다중 검증 레이어를 통해 처리한다. 자산별 이상 탐지, 시장 깊이 분석, 소스 간 분산 검증 등이 포함된다. 각 피드는 맞춤형 가격 집계를 거친 후, 담보를 보유한 다수의 독립 운영자들의 합의를 필요로 하는 합의 메커니즘(consensus mechanism)을 통해 최종적으로 검증된다.

높은 보안 및 데이터 품질 기준

레드스톤은 암호화폐 가격 피드를 위해 포괄적인 보안 아키텍처를 구현하고 있으며, 다계층 접근 방식을 통해 데이터 검증과 시스템 신뢰성을 보장한다. 이 인프라는 핵심 시스템 공학에서 차용한 원칙을 기반으로 구축되었으며, 데이터 무결성을 독립적으로 검증하는 분산 노드와 부분 장애 상황에서도 지속적인 운영을 보장하는 다중 중복 백업 시스템을 특징으로 한다. 엄격한 모니터링, 이상 탐지, 합의 기반 가격 형성, 데이터 제공자를 위한 경제적 인센티브를 통해 레드스톤은 출범 이후 단 한 번의 오가격(mispricing) 사례도 발생하지 않았다. 이러한 성과는 정기적인 제3자 보안 감사, 지속적인 자동화 테스트, 그리고 가격 데이터 품질에 영향을 미치지 전에 잠재적 이상 징후에 신속하게 대응할 수 있도록 하는 정교한 경보 시스템을 포함하는 견고한 개발 수명주기에 의해 뒷받침된다. 레드스톤의 코드는 여러 차례 보안 감사를 받았으며, 최상위 디파이 프로토콜들이 신뢰하고 있다.

시작하기

필요에 따라 레드스톤(RedStone)은 다양한 방식으로 통합할 수 있다.

푸시(Push) 모델: 디파이 상호운용성을 위한 표준화된 접근

기존 프로토콜 및 전통적인 데이터 소비 패턴과 깊이 통합된 dApp에 이상적이다.

레드스톤의 푸시 모델은 전통적인 블록체인 오라클 방식에 따라 주기적인 업데이트를 통해 데이터를 온체인에 직접 저장한다. 이 모델은 지속적인 데이터 가용성을 보장하며 애플리케이션이 업데이트 조건과 데이터 소스를 완전히 제어할 수 있도록 한다.

여기에서 업데이트 매개변수와 함께 제공되는 **모든 피드 목록**을 확인할 수 있다.

info

레드스톤 푸시는 기존의 푸시 오라클에 비해 상당한 장점을 지닌다. 모듈형 설계 덕분에 가격이 언제, 어떻게 업데이트될지에 대해 결정적인 권한을 갖게 되며(다른 오라클의 경우 정해진 매개변수를 그대로 수용해야 함), 사용자가 직접 선택할 수 있다.

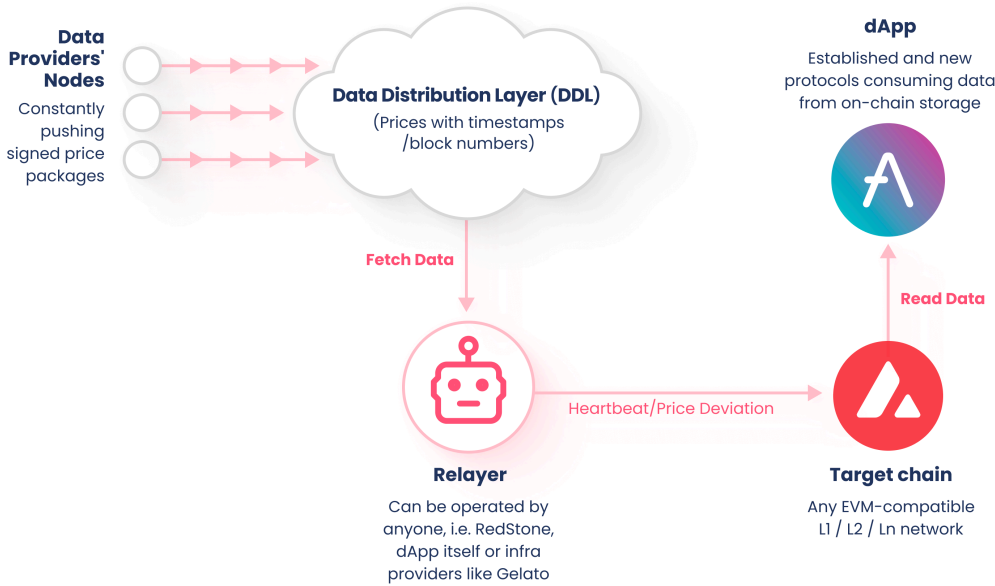
레드스톤 푸시 작동 방식

이 접근 방식은 **레드스톤 풀(Pull)** 모델을 기반으로 하며, 데이터 제공자와 타임스탬프에 대한 온체인 검증의 보안을 유지한다.

이 모델은 두 가지 주요 구성 요소로 이루어져 있다. 첫 번째는 **환경 변수**를 사용해 맞춤형 방식으로 데이터를 온체인에 푸시하는 역할을 하는 오프체인 **릴레이어(relayer)**이다. 두 번째는 가격을 저장하고 익숙한 인터페이스(예: **체인링크 애그리게이터**)를 통해 이를 가져올 수 있도록 하는 온체인 **컨트랙트(contracts)**이다. 레드스톤 푸시는 모든 EVM 호환 L1 및 L2, 스타크넷(Starknet), 퓨얼 네트워크(Fuel Network)에서 사용할 수 있다.

RedStone Push

Ideal for: Blue chip protocols introducing new assets (i.e. uniquely delivered by RedStone) and expanding to new networks, with no code changes needed



릴레이어(Relayer)

릴레이어는 **환경 변수**를 기반으로 맞춤형으로 동작하는 서비스이다. 정의된 조건을 주기적으로 검사해 조건 충족 시 가격을 푸시한다. UPDATE_CONDITIONS에 여러 조건을 전달할 수 있으며, 이 경우 릴레이어는 조건 중 하나라도 충족되면 가격을 업데이트하는 방식으로 작동한다. 현재 구현된 조건은 두 가지다:

- UPDATE_PRICE_INTERVAL 변수(밀리초 단위)로 정의된 time 조건: 가격이 얼마나 자주 업데이트되어

야 하는지를 지정(하트비트 개념), [코드](#)

- `MIN_DEVIATION_PERCENTAGE` 변수로 정의된 `value-deviation` 조건: 가격 업데이트를 트리거하기 위해 값이 얼마나 변동해야 하는지를 지정, [코드](#)

info

릴레이어는 무허가(permissionless) 방식으로 누구나 운영할 수 있으며, 데이터는 프로토콜 이해관계자가 정의한 조건에 따라 결국 온체인에서 검증된다. 또한 릴레이어는 병렬로 동작하도록 설계되어 있으며, 단일 장애 지점(SPoF)이나 검열 위험을 줄이기 위해 다수의(이상적으로 독립적인) 인스턴스를 운영하는 것이 권장된다.

컨트랙트(Contracts)

온체인 릴레이어는 [PriceFeedsAdapter](#) 컨트랙트를 기반으로 하며, 다음 기능을 담당한다:

- 모든 가격 피드의 심볼을 저장(RedStone dataFeedId로 매핑),
- 가격 피드 값을 저장,
- 가격 피드 값을 일괄 업데이트,
- 마지막 업데이트의 라운드 번호와 타임스탬프에 대한 정보 저장,
- 한 번의 호출로 여러 피드의 가격 값을 가져오기.

추가적으로, 프로토콜이 체인링크(Chainlink) PriceFeed 아키텍처와 100% 호환되기를 원한다면, 이를 모방하는 [PriceFeed](#) 컨트랙트를 배포하는 것도 가능하다.

환경 변수(Environment variables)

변수명(Variable)	설명(Description)
RELAYER_ITERATION_INTERVAL	릴레이어가 가격 업데이트를 시도하는 시간 간격
UPDATE_CONDITIONS	가격 업데이트 가능 여부를 결정하는 매개변수 배열, 현재 허용되는 매개변수는 'time'과 'value-deviation'
UPDATE_PRICE_INTERVAL	UPDATE_CONDITIONS에 "time" 매개변수가 포함된 경우, 가격이 얼마나 자주 업데이트되어야 하는지를 설명하는 시간 간격
MIN_DEVIATION_PERCENTAGE	UPDATE_CONDITIONS에 "value-deviation"이 포함된 경우, 가격 업데이트를 트리거하는 최소 가격 편차
RPC_URL	블록체인과 상호작용하기 위한 RPC URL
CHAIN_NAME	릴레이어가 작동해야 하는 블록체인의 이름
CHAIN_ID	릴레이어가 작동해야 하는 블록체인의 체인 ID
PRIVATE_KEY	어댑터 컨트랙트에 가격을 푸시하기 위해 올바른 네트워크에 자금을 보유한 지갑의 개인 키
ADAPTER_CONTRACT_ADDRESS	올바른 네트워크에 배포된 어댑터 컨트랙트의 주소
DATA_SERVICE_ID	어떤 데이터 서비스를 사용해 가격을 가져올지를 설명하는 레드스톤 래퍼(Wrapper) 매개변수
UNIQUE_SIGNERS_COUNT	가격 데이터에 서명해야 하는 고유 서명자의 수를 설명하는 레드스톤 래퍼 매개변수
DATA_FEEDS	어떤 토큰들이 사용될지를 설명하는 레드스톤 래퍼 매개변수
CACHE_SERVICE_URLS	가격을 가져오기 위해 사용될 캐시 서비스 URL을 설명하는 레드스톤 래퍼 매개변수
GAS_LIMIT	가격 피드 컨트랙트에 데이터를 푸시할 때 사용되는 가스 한도

풀(Pull) 모델: 다양한 소스로부터의 실시간 데이터

레드스톤의 풀 모델은 데이터를 사용자 트랜잭션에 직접 삽입하여 가스 효율성을 극대화하고 dApp의 데이터 접근을 단순화한다.

이 간소화된 접근 방식은 전체 과정을 단일 트랜잭션으로 처리해 비용을 크게 줄여준다.

낮은 지연(latency)의 데이터가 필요한 dApp에 가장 적합하다.

설치(Installation)

NPM 레지스트리에서 [@redstone-finance/evm-connector](#)와 [@redstone-finance/sdk](#)를 설치한다.

하드햇(Hardhat)

Yarn

```
yarn add @redstone-finance/evm-connector @redstone-finance/sdk
```

npm

```
npm install @redstone-finance/evm-connector @redstone-finance/sdk
```

파운드리(Foundry)

파운드리에는 Git 서브모듈을 사용하여 의존성을 설치하므로, 추가적인 [의존성 설치](#) 단계가 필요하다.

파운드리 프로젝트에서:

1. [@redstone-finance/evm-connector](#) 설치 - 메인 브랜치의 최신 코드를 설치한다.

```
forge install redstone-finance/redstone-oracles-monorepo
```

2. [@OpenZeppelin](#) 컨트랙트 설치 ([@redstone-finance/evm-connector](#)의 의존성) - 메인 브랜치의 최신 코드를 설치한다.

```
forge install OpenZeppelin/openzeppelin-contracts@v4.9.5
```

3. `remappings.txt`에 라이브러리 추가

```
echo "@redstone-finance/evm-connector/dist/contracts/=lib/redstone-oracles-monorepo/packages/evm-connector/contracts/
@openzeppelin/contracts=lib/openzeppelin-contracts/contracts/" >>
remappings.txt
```

사용법(Usage)

TLDR;

두 가지 작업이 필요하다.

1. [스마트 컨트랙트 수정](#) — 데이터 추출 및 검증을 담당하는 라이브러리를 포함하도록 스마트 컨트랙트를 조정한다.
2. [dApp의 자바스크립트 코드 수정](#) — 데이터 피드가 포함된 추가 페이로드를 주입해야 하며, 그렇지 않으면 스마트 컨트랙트 오류가 발생한다.

1. 스마트 컨트랙트 조정(Adjust your smart contracts)

Heads up

1. 레드스톤 컨트랙트는 `solidity > 0.8.4` 버전을 요구한다. 코드가 더 오래된 버전으로 작성된 경우 수동 페이로드를 사용해야 한다.
2. 서드파티 애그리게이터를 사용하는 경우, 추가 페이로드 전달을 지원하는지 반드시 확인해야 한다.
3. Remix에서는 레드스톤을 테스트하지 말 것. Remix는 `evm-connector`가 수행하는 방식으로 트랜잭션을 수정하는 기능을 지원하지 않는다.
4. 컨트랙트에는 업그레이드 메커니즘을 두는 것이 강력히 권장된다(멀티시그 또는 DAO 기반 가능). 이렇게 하면 문제 발생 시 데이터 제공자를 신속히 교체할 수 있다.

스마트 컨트랙트가 데이터에 접근할 수 있도록 소스 코드에 최소한의 변경을 적용해야 한다. 사용하는 데이터 서비스에 따라 컨트랙트는 레드스톤의 기본 컨트랙트(base contracts) 중 하나를 상속해야 한다.

데이터 서비스별 기본 컨트랙트 목록

기본 컨트랙트(Base Contract)	데이터 서비스 (피드 목록 포함)	상태(Status)
MainDemoConsumerBase.sol	redstone-main-demo	Demo
AvalancheDataServiceConsumerBase.sol	redstone-avalanche-prod	Production
PrimaryProdDataServiceConsumerBase.sol	redstone-primary-prod	Production



참고: *Production* 상태의 서비스는 다수의 노드가 배포되어 있으며, 전문적으로 모니터링된다.

```
import "@redstone-finance/evm-connector/contracts/data-  
services/MainDemoConsumerBase.sol";  
  
contract YourContractName is MainDemoConsumerBase {  
    ...  
}
```

데이터 피드 ID는 `bytes32` 형식으로 변환해 전달한다.

단일 값 가져오기

```
uint256 ethPrice = getOracleNumericValueFromTxMsg(bytes32("ETH"));
```

여러 값 가져오기

```
bytes32 ;
dataFeedIds[0] = bytes32("ETH");
dataFeedIds[1] = bytes32("BTC");
uint256[] memory values = getOracleNumericValuesFromTxMsg(dataFeedIds);
uint256 ethPrice = values[0];
uint256 btcPrice = values[1];
```

지원되는 모든 피드에 대해서는 [차트와 이력 데이터를 포함한 UI](#)를 제공한다.

💡 참고: 다음 함수들은 오버라이드할 수 있다(사용자 책임 하에):

- `isTimestampValid(uint256 receivedTimestamp)` returns (bool)
타임스탬프 검증 로직을 커스터마이징할 수 있다. 최신 가격만 허용하도록 짧은 지연 시간을 지정할 수도 있고, 블록 시간이 긴 네트워크에서는 너무 많은 트랜잭션이 거부되지 않도록 이 기간을 늘릴 수도 있다.
- `aggregateValues(uint256[] memory values)` returns (uint256)
여러 공급자의 값을 집계하는 로직을 커스터마이징할 수 있다(기본적으로는 중앙값 사용). 예를 들어, 불연속 값 처리 시 공급자 값이 모두 동일해야 한다고 지정할 수 있다.
- `getAuthorisedSignerIndex(address _signerAddress)` returns (uint256)
추가 서명자를 화이트리스트에 등록하거나, 손상된 서명자를 제거할 수 있다.
- `getUniqueSignersThreshold()` returns (uint256)
필요한 서명자 수를 변경할 수 있다. 값이 높을수록 신뢰성은 커지지만 가스 비용도 증가한다.

2. dApp의 자바스크립트 코드 조정(Adjust Javascript code of your dApp)

트랜잭션 전송을 담당하는 코드도 업데이트해야 한다. [ethers.js\(이더스.js\)](#)를 사용 중이라면, 전환을 매끄럽게 하기 위한 전용 라이브러리를 제공한다.

컨트랙트 객체 래핑(Contract object wrapping)

먼저, 프로젝트에 래퍼(wrapper) 코드를 임포트한다.

JavaScript

```
const { WrapperBuilder } = require("@redstone-finance/evm-connector");
```

TypeScript

```
import { WrapperBuilder } from "@redstone-finance/evm-connector";
```

그런 다음 선택한 [RedStone 데이터 서비스 ID](#)를 지정해 ethers 컨트랙트를 래핑할 수 있다. 선택적으로 고유 서명자 수, 데이터 피드 식별자, RedStone 캐시 노드의 URL을 지정할 수 있다.

```
import { getSignersForDataServiceId } from "@redstone-finance/sdk";

const yourEthersContract = new ethers.Contract(address, abi, provider);

const wrappedContract = WrapperBuilder.wrap(contract).usingDataService({
  dataPackageIds: ["ETH", "BTC"],
  authorizedSigners: getSignersForDataServiceId(DATA_SERVICE_ID),
});
```

이제 ethers-js 코드로 상호작용하듯 동일한 방식으로 컨트랙트의 모든 메서드에 접근할 수 있다.

```
wrappedContract.executeYourMethod();
```

테스트(Testing)

Hardhat(하드햇)

테스트 컨텍스트에서 래퍼를 사용하려면, 서로 다른 시나리오를 쉽게 검증할 수 있도록 오라클 값을 오버라이드할 수 있는 모의(mock) 래퍼 사용을 권장한다. 모의 래퍼를 사용하려면 `usingDataService` 대신 `usingMockData(signedDataPackages)` 함수를 사용한다.

```
const {
  SimpleNumericMockWrapper,
} = require("@redstone-finance/evm-connector/dist/src/wrappers/SimpleMockNumericWrapper");

const wrappedContract = WrapperBuilder.wrap(
  yourContract,
).usingSimpleNumericMock({
  mockSignersCount: 10,
  dataPoints: [{ dataFeedId: "ETH", value: 1000 }],
});
await wrappedContract.yourMethod();
```

모킹 데이터의 더 많은 예시는 [여기](#)에서 확인할 수 있다.

Foundry(파운드리)

Foundry 테스트 컨텍스트에서 RedStone을 사용하려면, 모의 dataPackages 생성을 위해 foundry의 `vm.ffi` 함수를 사용할 것을 권장한다. RedStone과 Foundry를 통합하는 방법을 보여주는 [레포지토리](#)를 준비했다.

- [Foundry 컨트랙트에서 RedStone 페이로드 소비 예시](#)
- [RedStone 모의 페이로드 생성 스크립트](#)

수동 페이로드(Manual payload)

이 접근 방식은 프로토콜 내에서 한 컨트랙트에서 다른 컨트랙트로 가격 데이터를 전달해야 하는 경우에 유용하다.

또한 컨트랙트가 0.8.4보다 낮은 Solidity 버전으로 작성되어 RedstoneConsumerBase 컨트랙트를 상속하기 어렵다면, 검증 로직을 포함하는 별도의 Extractor 컨트랙트를 배포하는 해법이 된다.

```
pragma solidity 0.8.4;

import "@redstone-finance/evm-connector/contracts/mocks/RedstoneConsumerNumericMock.sol";

contract RedstoneExtractor is RedstoneConsumerNumericMock {

    function extractPrice(bytes32 feedId, bytes calldata redstonePayload)
    public view returns(uint256) {
        return getOracleNumericValueFromTxMsg(feedId);
    }
}
```

그리고 원본 컨트랙트에서 페이로드를 프록시한다.

```
function getPriceFromRedstoneOracle(bytes32 feedId, bytes calldata
redstonePayload) public view returns(uint256) {
    return redstoneExtractor.extractPrice(feedId, redstonePayload);
}
```

클라이언트 측에서는 다음 코드를 사용해 수동 페이로드를 생성할 수 있다.

```
const redstonePayload = await new DataServiceWrapper({
    dataServiceId: "redstone-main-demo",
    dataFeeds: ["ETH"],
}).getRedstonePayloadForManualUsage(yourContract);

// 컨트랙트와 상호작용(오라클 값을 안전하게 가져오기)
const price = await yourContract.getPriceFromRedstoneOracle(redstonePayload);

## 작동 데모(Working demo)
```

@redstone-finance/evm-connector 사용 예시는 우리의 [예제 전용 리포지토리](#)에서 확인할 수 있다.

레드스톤 SDK(RedStone SDK)

레드스톤 데이터를 사용하는 가장 일반적인 방식은 온체인으로 전달하는 것이지만, 오프체인에서 소비하고 싶을 수도 있다.

이 시나리오를 위해 [@redstone-finance/sdk](#)를 제공한다.

두 모델의 장점 결합(Best of two models)

푸시(Push)와 풀(Pull) 모델의 결합

이 모델은 [ERC7412](#) 형태로 도입되었으며, 구현 전에 문서를 읽을 것을 권장한다. 해당 모델은 파페추얼 프로토콜인 [Synthetix](#)에 의해 대중화되었다.

important Requirements

TLDR; 다음 두 가지가 필요하다.

1. 가격 피드 배포(Deploy price feed)
2. 클라이언트 코드를 [erc7412](#) 사용으로 수정

가이드(Guide)

가격 피드 컨트랙트 배포(Deploy price feed contract)

1. 의존성 설치: `npm install @redstone-finance/erc7412`
2. `@redstone-finance/erc7412/contracts/RedstoneERC7412.sol`에서 가져오는 `RedstonePrimaryProdWithoutRoundsERC7412` 컨트랙트를 상속해야 한다.
 1. `getTTL` 메서드를 구현한다. 이 메서드는 컨트랙트 내 가격이 오래됨(stale) 상태가 되는 데 걸리는 시간을 초(초 단위)로 반환해야 한다. 가격이 오래되면 피드 컨트랙트는 업데이트 전까지 읽기 시에 `revert`된다. 가격 업데이트는 아래 "Modify DAPP" 절에서 설명된 방식으로 수행된다.
 2. 배포할 피드에 대한 `dataFeedId` 를 선택한다. 전체 [지원 자산 목록](#)을 참고한다.
3. 컨트랙트를 배포한다.

BTC dataFeedId용 예시 컨트랙트

```
import {RedstonePrimaryProdWithoutRoundsERC7412} from '@redstone-finance/erc7412/contracts/RedstoneERC7412.sol';

contract BTCFeed is RedstonePrimaryProdWithoutRoundsERC7412 {
    function getTTL() override view internal virtual returns (uint256) {
        return 3600;
    }

    function getDataFeedId() override view public virtual returns (bytes32) {
        return bytes32("BTC");
    }
}
```

DAPP 수정(Modify DAPP)

dApp은 `erc7412` 표준을 지원해야 하며, 피드 내 가격이 오래된(stale) 경우 사용자가 가격을 업데이트할 수 있도록 해야 한다.

참고: 사용자가 가격을 업데이트해야 하는 상황이 발생하면, 해당 가스 트랜잭션 비용을 추가로 지불해야 한다.

dApp 수정은 사용자가 트랜잭션을 실행하기 직전에 호출해야 하는 `generate7412CompatibleCall` 함수 추가를 요구한다.

1. 현재 `erc7412` 라이브러리는 `viem` 클라이언트에 의존한다.
2. 먼저 컨트랙트 호출 데이터를 준비하고, 다음 단계에서 이를 `generate7412CompatibleCall`에 전달한다.

```
import { generate7412CompatibleCall } from "@redstone-
finance/erc7412/generate7412CompatibleCall";

// 사용자 컨트랙트 함수 호출을 인코딩
const callData = viem.encodeFunctionData({
  functionName: "your contract function",
  args: [],
  abi: your_contract_abi,
});

// 이 함수는 트랜잭션을 시뮬레이션한다.
// 만약 erc7412.OracleDataRequired 때문에 실패하면,
// RedStone 게이트웨이에서 오라클 페이로드를 가져와
// {user_tx, update_redstone_price_feed_tx} 두 트랜잭션으로 구성된 멀티콜 트랜잭션을 준
비한다.
const call = await generate7412CompatibleCall(
  await hardhat.viem.getPublicClient(),
  btcPriceFeed.address,
  wallet.account.address,
  callData,
  multicall.address,
);

// 트랜잭션 전송
// 먼저 가격 피드를 업데이트한 뒤
// 변경되지 않은 사용자 트랜잭션을 실행한다.
await wallet.sendTransaction(call);

// 데이터가 이미 컨트랙트에 설정되었으므로 TTL이 지날 때까지 업데이트가 필요 없다.
console.log("BTC price:", await btcPriceFeed.read.latestAnswer());
```

작동 예시는 [여기](#)에서 확인할 수 있다.

비EVM 체인(Non-EVM chains)

레드스톤은 몇몇 비EVM 호환 체인도 지원한다.

솔라나(Solana)

레드스톤 솔라나 가격 어댑터(RedStone Solana Price Adapter)

- [PriceAdapter](#) 컨트랙트는 하나 이상의 오프체인 프로세스에 의해 업데이트된다.
- 데이터는 피드 ID마다 생성된 계정에 [PriceData](#) 구조체로 기록된다.
- 컨트랙트는 view 함수들 중 하나로 읽을 수 있다.
- 매개변수/로직 업데이트는 코드 변경을 필요로 한다.
 - 다만, **가격 피드 계정 주소(Price Feed Account Address)**는 변경되지 않으며 영구적으로 읽을 수 있다.
- IDL 파일은 [모노레포 디렉토리](#)에서 확인할 수 있다.
 - 여기서 address 필드는 비어 있으므로, 배포된 주소를 사용해야 한다.

이미 배포된 주소들

- 메인넷(Mainnet): [REDSTBDUecGjwXd6YGPzHSvEUBH0qVRfCcJUVgPiHsr](#)
- 테스트넷(Testnet): [rds8J7VKqLQgzDr7vS59dkQga3B1BotgFy8F7LSLC74](#)
- 개발망(Devnet): [REDuYsnEucMweattdv4xQCYdU1i8Q2W92kdayrpY9rA](#)

! 테스트넷 및 개발망 컨트랙트는 레드스톤 개발 인프라를 기반으로 배포되었으며, 업그레이드 시 일시적인 데이터 누락이 발생할 수 있다.

레드스톤 가격 피드 계정(RedStone Price Feed Account)

구조(Structure)

각 가격 피드 계정의 구조는 다음과 같이 정의된다:

```
#[account]
pub struct PriceData {
    pub feed_id: [u8; 32], // 피드 ID ASCII 바이트를 오른쪽 0으로 채운 바이트 배열
    pub value: [u8; 32], // BigEndian으로 인코딩된 바이트 배열 (u256 값, 10^decimals
로 곱해진 상태)
    pub timestamp: u64, // 밀리초 단위 - 통합된 레드스톤 데이터 타임스탬프
    pub write_timestamp: Option<u64>, // 밀리초 단위 - 데이터가 기록된 슬롯의 타임스탬프
    pub update_slot: u64, // 데이터가 기록된 슬롯 번호
    pub decimals: u8, // 현재 레드스톤 가격의 경우 8로 고정
    _reserved: [u8; 64] // 예약된 바이트
}
```

읽기(Reading)

역직렬화(Deserializing)

다음 제너릭 메서드는 모든 솔라나 계정을 PriceData로 역직렬화하는 데 사용할 수 있다:

```
use anchor_lang::prelude::*;
use anchor_lang::Discriminator;

pub fn account_deserialize<T: AccountDeserialize + Discriminator>(
    account: &AccountInfo<'_,
) -> T {
    let data = account.clone().data.borrow().to_owned();
    let discriminator = data.get(..8).unwrap();
    if discriminator != T::discriminator() {
        panic!(
            "Expected discriminator for account {:?} ({:?}) is different from
received {:?}",
            account.key(),
            T::discriminator(),
            discriminator
        );
    }
    let mut data: &[u8] = &data;

    T::try_deserialize(&mut data).unwrap()
}
```

가격 값 읽기(Reading Price Value)

가격 값은 길이 32의 BigEndian 바이트 리스트로 인코딩된다.

아래는 u64로 변환하는 예시이며, 큰 값에서는 오버플로우가 발생할 수 있다:

```
fn redstone_value_to_price(raw_be_value: [u8; 32]) -> Result<u64> {
    if !raw_be_value.iter().take(24).all(|&v| v == 0) {
        warn!("Price overflow u64");
        return Err(...); // 오버플로우
    }

    u64::from_be_bytes(raw_be_value[24..].try_into().unwrap())
}
```

⚠ 값은 10^{decimals} 로 곱해져 있다는 점을 기억해야 한다.

사용 예시(Example usage)

```
#[program]
```

```
pub mod test_usage {
    use super::*;
    pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
        let price_data: PriceData =
            account_deserialize(&ctx.accounts.price_info);

        let value = redstone_value_to_price(price_data.value);
        msg!("FeedId: {:?}, {:?}" , price_data.feed_id, value);

        Ok(())
    }
}

#[derive(Accounts)]
pub struct Initialize<'info> {
    /// CHECK: ...
    #[account()]
    pub price_info: AccountInfo<'info>,
}
```

계정 주소 유도(Deriving Account Addresses)

계정의 공개키를 유도하려면 아래 코드 스니펫을 사용할 수 있다.

Rust

"price"라는 상수 문자열과 feed_id를 시드로 사용한다:

```
type FeedIdBs = [u8; 32]; // 피드 ID ASCII 바이트를 오른쪽 0으로 채운 바이트 배열
const PROGRAM_ID: &str = "REDSTBDUecGjwXd6YGPzHSvEUBHQqVRfCcjUVgPiHsr";

fn make_price_seed() -> [u8; 32] {
    let mut seed = [0u8; 32];
    seed[0..5].copy_from_slice(b"price");

    seed
}

fn price_feed_account_pubkey(feed_id: &FeedIdBs) -> (Pubkey, u8) {
    Pubkey::find_program_address(&[
        &make_price_seed(), feed_id
    ], &Pubkey::from_str(PROGRAM_ID).unwrap())
}
```

TypeScript

```
const PROGRAM_ID = new
PublicKey("REDSTBDUecGjwXd6YGPzHSvEUBHQqVRfCcjUVgPiHsr");
```

```
const FEED_ID = "ETH";

const makeFeedIdBytes = (feedId: string) => {
  return Buffer.from(feedId.padEnd(32, "\0"));
};

const makePriceSeed = () => {
  return Buffer.from("price".padEnd(32, "\0"));
};

const seeds = [makePriceSeed(), makeFeedIdBytes(FEED_ID)];

const address = PublicKey.findProgramAddressSync(seeds, PROGRAM_ID)[0];
```

가격 피드 계정 목록(Price Feed Account List)

기존 가격 피드 계정 목록은 다음에서 확인할 수 있다:

<https://github.com/redstone-finance/redstone-oracles-monorepo/tree/main/packages/solana-connector/deployments/solanaMultiFeed/programs#feed-accounts>

라디스(Radix) — 시작하기

1. Get Started 섹션에서 레드스톤 모델을 읽는다.
2. @redstone-finance/radix-connector 랜딩 페이지를 따른다.

Rust/Scrypto에서 사용법

Scrypto 컴포넌트 — 구성 방식

1. 먼저 [How to start](#) 섹션의 문서를 읽는다.
2. [로컬 Scrypto 환경 설정](#) 방법을 확인한다.
3. [온레저 컴포넌트의 일반적 철학](#)을 읽는다.
4. 컴포넌트의 전체 소스는 [여기](#)에서 확인할 수 있다.
5. 컴포넌트가 기반한 [RedStone Rust SDK](#) 문서를 확인한다.

 여기에 기술된 정보는 아래에 대부분 반복되지 않는다.

의존성(Dependencies)

RedStone Rust SDK를 Scrypto에 임베드하려면 다음 의존성을 사용한다:

```
[dependencies]
scrypto = { version = "1.3.0" }
redstone = { git = "https://github.com/redstone-finance/rust-sdk", tag =
"2.0.0", default-features = false, features = ["radix"] }
```

인스턴스화 및 저장(Instantiating & Storage)

일부 값은 컴포넌트 인스턴스화 과정에서 저장해야 한다.

또한 푸시 모델의 경우, 값과 타임스탬프가 컴포넌트에 저장된다.

```
#[blueprint]
mod price_adapter {
    struct PriceAdapter {
        signer_count_threshold: u8,
        signers: Vec<Vec<u8>>,
        prices: HashMap<Vec<u8>, RedStoneValue>,
        timestamp: u64,
    }

    impl PriceAdapter {
        pub fn instantiate(
            signer_count_threshold: u8,
            allowed_signer_addresses: Vec<Vec<u8>>,
        ) -> Global<PriceAdapter> {
            let addresses: Vec<SignerAddress> = allowed_signer_addresses
                .iter()
                .map(|signer| (signer.clone()).into())
                .collect();

            verify_signers_config(addresses.as_slice(),
signer_count_threshold)
                .unwrap_or_else(|err| panic!("{}", err));

            Self {
                signer_count_threshold,
                signers: allowed_signer_addresses,
                prices: hashmap!(),
                timestamp: 0,
            }
                .instantiate()
                .prepare_to_globalize(OwnerRole::None)
                .globalize()
        }
    }
}
```

RedStone Rust SDK 사용하기

페이로드 처리(Payload processing)

1. 페이로드 바이트는 [여기](#) 설명된 대로 정의해야 한다.
2. 페이로드는 [이 문서](#)에서 설명된 대로 생성할 수 있다.

다음 코드를 #[blueprint] 안에서 사용해 페이로드 데이터를 처리한다:

```
use redstone::{
    core::{config::Config, ProcessorResult},
    radix::RadixRedStoneConfig,
    Value as RedStoneValue,
    contract::verification::*,
    SignerAddress
};
```

해당 함수는 인자로 전달된 페이로드를 온체인에서 처리하고, feed_ids에 포함된 각 피드에 대한 집계 값 배열과 페이로드 데이터 패키지와 관련된 타임스탬프를 반환한다.

```
fn process_payload(
    &mut self,
    feed_ids: Vec<Vec<u8>>,
    payload: Vec<u8>,
) -> (u64, Vec<RedStoneValue>) {
    let result = Self::try_process_payload(
        Self::get_current_time(),
        self.signer_count_threshold,
        self.signers.clone(),
        feed_ids,
        payload,
        None,
        None,
    )
        .unwrap_or_else(|err| panic!("{}", err));

    (result.timestamp.as_millis(), result.values)
}

fn try_process_payload(
    block_timestamp: u64,
    signer_count_threshold: u8,
    signers: Vec<Vec<u8>>,
    feed_ids: Vec<Vec<u8>>,
    payload: Vec<u8>,
    max_timestamp_delay_ms: Option<u64>,
    max_timestamp_ahead_ms: Option<u64>,
) -> ProcessorResult {
    let config: RadixRedStoneConfig = Config::try_new(
        signer_count_threshold,
```

```

        signers.into_iter().map(|id| id.into()).collect(),
        feed_ids.into_iter().map(|id| id.into()).collect(),
        block_timestamp.into(),
        max_timestamp_delay_ms.map(|v| v.into()),
        max_timestamp_ahead_ms.map(|v| v.into()),
    )?
    .into();

    redstone::core::process_payload(&config, payload)
}

```

설정(Config)

Config 구조는 [여기](#)에 설명되어 있다.

보안상의 이유로, 허용된 signers와 signer_count_threshold는 위에서 정의된 대로 컴포넌트에 임베드해야 한다.

현재 타임스탬프(Current timestamp)

현재 타임스탬프(밀리초)는 block_timestamp 매개변수로 전달해야 한다:

```

use scrypto::prelude::*;

pub fn get_current_time() -> u64 {
    let rtn = ScryptoVmV1Api::object_call(
        CONSENSUS_MANAGER.as_node_id(),
        CONSENSUS_MANAGER_GET_CURRENT_TIME_IDENT,
        scrypto_encode(&ConsensusManagerGetCurrentTimeInputV2 {
            precision: TimePrecisionV2::Second,
        })
        .unwrap(),
    );

    let instant: Instant = scrypto_decode(&rtn).unwrap();

    instant.seconds_since_unix_epoch as u64
}

```

에러(Errors)

페이로드 처리 중 발생 가능한 에러는 [여기](#)에서 확인할 수 있다.

풀(Pull) 모델

풀 모델을 사용하려면 process_payload 함수를 호출하고 값을 반환하면 된다.

```
pub fn get_prices(
    &mut self,
    feed_ids: Vec<Vec<u8>>,
    payload: Vec<u8>,
) -> (u64, Vec<RedStoneValue>) {
    self.process_payload(feed_ids, payload)
}
```

푸시(Push) 모델

푸시 모델에서는 process_payload 함수를 호출하고 값을 스토리지에 저장한다.

```
pub fn write_prices(
    &mut self,
    feed_ids: Vec<Vec<u8>>,
    payload: Vec<u8>,
) -> (u64, Vec<RedStoneValue>) {
    let (payload_timestamp, values) = self.process_payload(feed_ids.clone(),
payload);

    UpdateTimestampVerifier::Untrusted.verify_timestamp(
        Self::get_current_time().into(),
        None,
        0.into(),
        self.timestamp.into(),
        payload_timestamp.into(),
    ).unwrap_or_else(|err| panic!("{}", err));

    self.timestamp = payload_timestamp;
    self.prices = feed_ids
        .iter()
        .zip(values.clone())
        .map(|(feed_id, value)| (feed_id.clone(), value))
        .collect();

    (payload_timestamp, values)
}
```

그런 다음 값은 다음과 같이 읽을 수 있다:

```
pub fn read_prices(&mut self, feed_ids: Vec<Vec<u8>>) -> Vec<RedStoneValue> {
    feed_ids
        .iter()
        .map(|feed_id| self.read_price(feed_id.clone()))
        .collect()
}
```

또는

```
pub fn read_price(&mut self, feed_id: Vec<u8>) -> RedStoneValue {
    *self.prices.get(&feed_id).unwrap()
}
```

TypeScript에서 사용법

1. 먼저 [How to start](#) 섹션의 문서를 읽는다.
2. TypeScript 계층에서 [컨트랙트/컴포넌트 연결](#) 방법을 확인한다.
3. 패키지의 전체 소스는 [여기](#)에서 확인할 수 있다.

⚠ 여기에 기술된 정보는 대부분 **반복되지 않는다**.

의존성(Dependencies)

다음 의존성을 사용해 *RedStone Radix Connector*를 TypeScript에 임베드한다:

```
{
  dependencies: {
    ...
    "@redstone-finance/radix-connector": "0.8.0",
    ...
  }
}
```

@redstone-finance/radix-connector로 PriceAdapter 패키지 사용하기

패키지 배포(Deploying package)

```
yarn sample-deploy
```

1. 위 명령어로 패키지를 배포할 수 있으며, 스크립트는 [여기](#)에 정의되어 있다.
2. 배포된 `package.stokenet.addr` 주소는 `scrypto/price_adapter` 디렉토리에서 확인할 수 있다.
3. 스크립트는 Radix Desktop Tool이 사용하는 [@atlantis-l/radix-tool](#) 패키지를 기반으로 하는

RadixPackageDeployer를 사용한다.

컴포넌트 인스턴스화(Instantiating component)

```
yarn sample-instantiate
```

1. 위 명령어로 컴포넌트를 인스턴스화할 수 있으며, 스크립트는 [여기](#)에 정의되어 있다.
2. 배포된 `component.tokenet.addr` 주소는 `scrypto/price_adapter` 디렉토리에서 확인할 수 있다.
3. 스크립트는 [PriceAdapterRadixContractDeployer](#)를 사용한다.

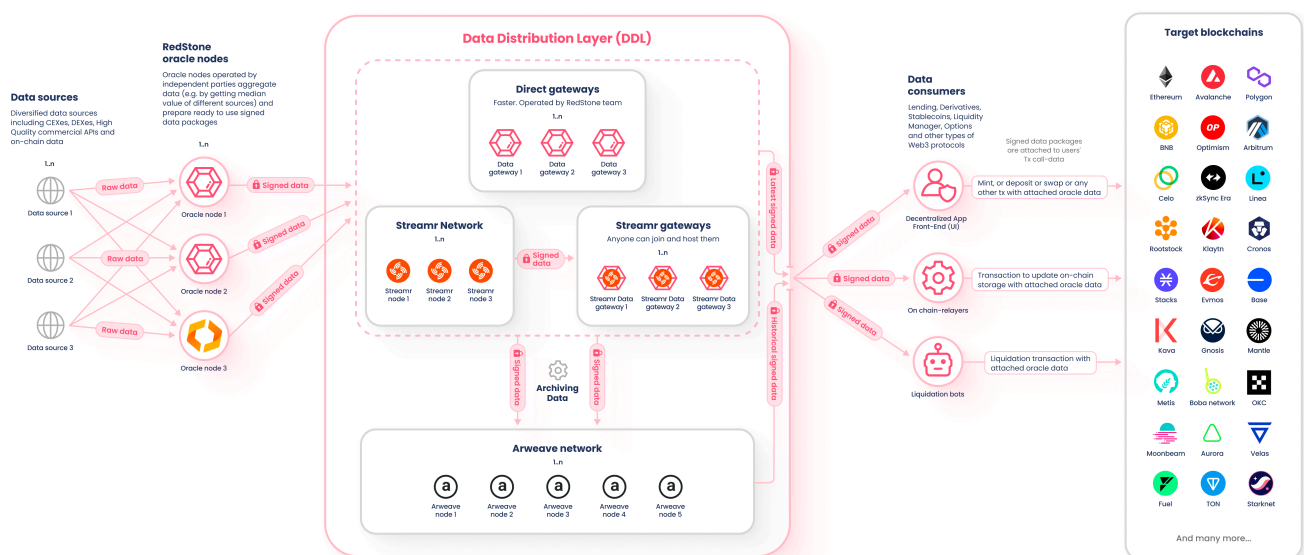
샘플 실행(Sample run)

```
yarn sample-run
```

1. 위 명령어로 샘플 시나리오를 실행할 수 있으며, 스크립트는 [여기](#)에 정의되어 있다.
2. 스크립트는 [PriceAdapterRadixContractConnector](#)를 사용한다.

기술 아키텍처(Technical architecture)

데이터가 블록체인으로 흐르는 방식(How Data Flows to the Blockchain)



개요(Overview)

레드스톤(RedStone)이 클라이언트에게 제공하는 가격 피드는 다양한 출처에서 수집된다. 여기에는 바이낸스(Binance), 코인베이스(Coinbase)와 같은 중앙화 거래소, 유니스왑(Uniswap), 스시스왑(Sushiswap)과 같은 탈중앙화 거래소(DEX), 코인마켓캡(CoinMarketCap), 코인게코(CoinGecko)와 같은 가격 집계기가 포함된다. 레드스톤은 현재까지 150개 이상의 출처를 통합했다. 데이터는 데이터 제공자가 운영하는 독립적인 노드에 의해 다양한 방법론으로 집계된다. 이 방법론에는 중앙값(median), TWAP(Time-Weighted Average Price), LWAP(Liquidity-Weighted Average Price)이 포함되며, 이는 모두 특정 기간 동안의 평균 가격, 유동성 규모 등을 고려해 가장 정확한 가격을 산출하도록 설계되었다.

추가적으로, 데이터 품질을 보장하기 위해 이상값 탐지(outlier detection)와 같은 품질 관리 절차가 적용된다. 이후 정제되고 처리된 데이터는 노드 운영자에 의해 서명되어 품질이 보증된다. 피드는 분산형 데이터 네트워크인 Streamr와, 필요 시 쉽게 분리하여 운영할 수 있는 오픈소스 게이트웨이를 통해 동시에 방송된다. 데이터는 특정 가격 변동 조건 등 사전에 정의된 조건 하에서 동작하는 전용 릴레이어나, 청산을 수행하는 봇, 또는 프로토콜과 상호 작용하는 최종 사용자에게 의해 블록체인에 푸시될 수 있다. 프로토콜 내부에서는 데이터가 언패킹(unpacking)되고, 출처와 타임스탬프가 올바른지 암호학적으로 검증된다.

데이터 포매팅 및 처리(Data Formatting & Processing)

맥락(Context)

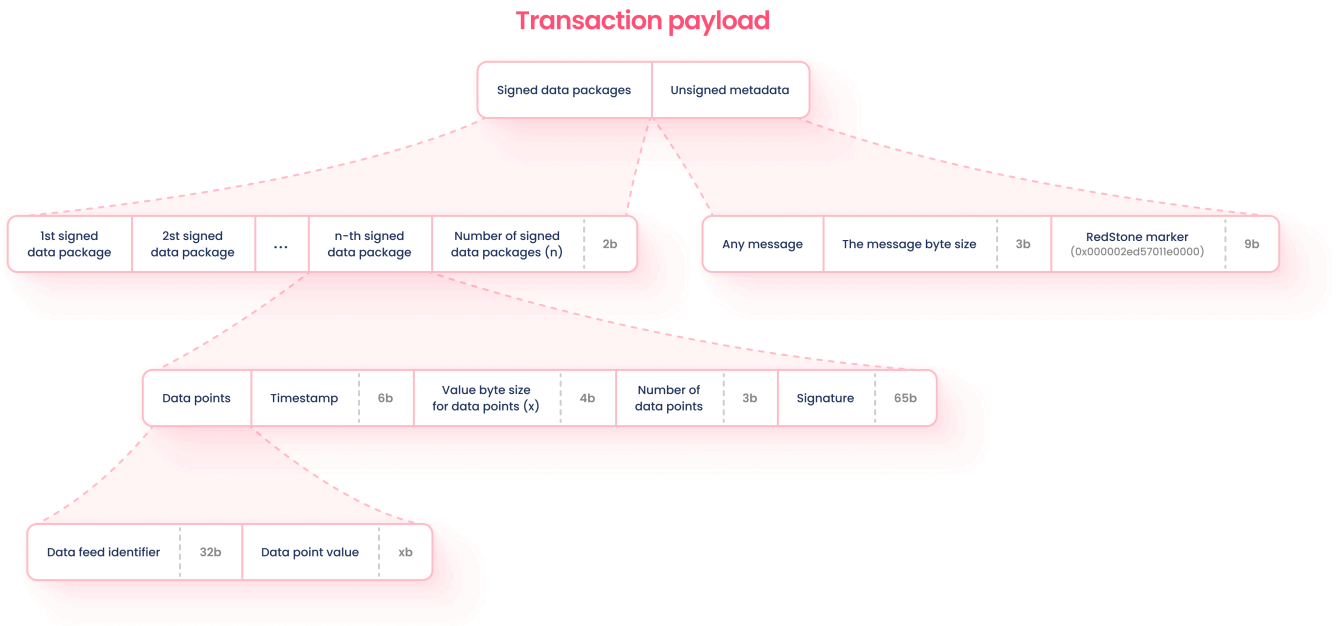
데이터를 블록체인으로 전송하려면 사용자 트랜잭션에 추가 페이로드를 패키징하고, 블록체인에서 메시지를 처리해야 한다. 다시 말해, 암호화해 가격과 같은 데이터는 사용자의 트랜잭션을 구성하는 데이터 일부에 삽입된다. 이는 블록체인이 상태(state)에서 상태로 이동하며 콜 데이터(call data)를 포함하기 때문에 가능하다. 레드스톤은 이러한 콜 데이터에 자체 데이터를 삽입함으로써, 블록체인에 데이터를 기록할 수 있다.

블록체인에 데이터가 기록되기 전 인코딩 방식(How Data is Encoded Before Being Put on the Blockchain)

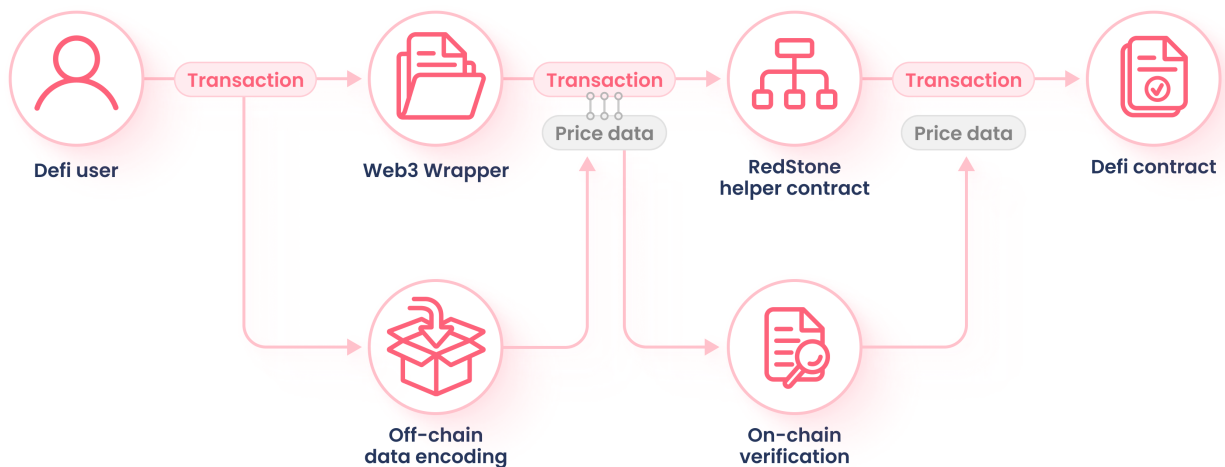
note

모든 단계는 ContractWrapper에 의해 자동으로 실행되며 최종 사용자에게는 투명하다.

1. 필요한 데이터를 Streamr 네트워크 또는 레드스톤 게이트웨이로 구동되는 데이터 배포 계층에서 가져온다.
2. 데이터를 아래 '트랜잭션 페이로드(Transaction Payload)' 다이어그램에 따라 메시지로 패키징한다.



3. 패키지를 원본 트랜잭션 메시지에 추가하고, 서명한 뒤 네트워크에 제출한다.



온체인 데이터 언패킹·검증·집계 방식(How Data Is Unpacked, Verified and Then Aggregated On-Chain)

먼저, 추가된 데이터 패키지가 콜 데이터에서 추출된다. 이후 보안 절차가 수행되며, 신뢰할 수 있는 제공자가 생성한 서명인지 검증하고, 타임스탬프를 확인해 정보가 올바른지 점검한다. 이후 요청된 각 데이터 피드마다 레드스톤은 수신된 고유 서명자의 수를 계산하고, 각 고유 서명자의 값을 추출한 뒤 이를 집계한다. 기본적으로 제공되는 값은 모든 값의 중앙값(median)이다. 이 로직은 온체인 환경에서 실행되며, 가스 소비를 최소화하기 위해 저수준 어셈블리 코드로 최적화되었다.

레드스톤 오라클 시스템의 보안을 강화하기 위해 온체인 집계 메커니즘을 도입했다. 이 메커니즘은 최소한의 고유 데이터 피드 수를 확보해야 한다는 추가 요건을 적용한다. 서로 다른 공급자로부터 제공된 값은 소비자 컨트랙트로 반환되기 전에 집계된다. 기본적으로 레드스톤은 중앙값 계산을 사용한다. 이렇게 하면 일부 공급자가 손상되더라도 (예: 10명 중 2명), 집계 값에 큰 영향을 미치지 않는다.

레드스톤 컨슈머 기본 컨트랙트의 온체인 집계 파라미터(On-Chain Aggregation Parameters in RedStone's Consumer Base Contract)

- `getUniqueSignersThreshold` 함수
목적: 데이터를 검증하기 위해 필요한 고유 서명자의 임계값을 결정한다. 레드스톤은 정확성과 무결성을 보장하기 위해 다수의 독립 서명자에 의존한다.
- `getAuthorisedSignerIndex` 함수
목적: 서명자 목록에서 승인된 서명자의 인덱스를 반환한다. 특정 서명자가 데이터에 서명할 권한이 있는지 검증하는 데 사용된다.
- `aggregateValues` 함수 (숫자 값)
목적: 여러 데이터 포인트에서 숫자 값을 집계한다. 중앙값 같은 평균을 계산할 수 있다. 다수 출처의 값을 집계하면 단일 오류 데이터 포인트의 영향을 줄일 수 있다.
- `aggregateByteValues` 함수 (바이트 배열)
목적: 바이트 배열에 특화된 값 집계를 수행한다.

지원되는 값 유형(Types of Values Supported)

컨트랙트에서 수신할 수 있는 데이터 유형은 다음 두 가지다:

1. 숫자형 256비트 값 (기본값)
2. 가변 크기의 바이트 배열

보안 고려사항(Security Considerations)

- `getUniqueSignersThreshold`를 오버라이드하는 것은 상당한 위험이 있으며, 100% 확신이 있는 경우에만 권장된다.
- 타임스탬프 검증 로직에 주의해야 한다. 일부 사용 사례(예: 합성 DEX)에서는 최신 값을 컨트랙트 스토리지에 캐싱해 차익거래 공격을 방지해야 한다.
- 컨트랙트에 보안 업그레이드 메커니즘을 반드시 포함시켜야 한다(멀티시그 또는 DAO 기반이 이상적).
- 레드스톤 데이터 서비스 레지스트리를 모니터링하고, 변경 시 컨트랙트에서 서명자 승인 로직을 신속히 수정해야 한다(유료 클라이언트에게는 별도 알림 제공).

권장사항(Recommendations)

- 스마트 컨트랙트를 설계할 때, 동일한 트랜잭션에서 너무 많은 데이터 피드를 요청하지 않도록 한다.
- 약 3명의 고유 서명자를 요구하는 것이 보안성과 가스 비용 최소화 사이에서 권장되는 균형이다.

벤치마크(Benchmarks)

벤치마크 스크립트와 보고서는 [여기](#)에서 확인할 수 있다.

데이터 품질(Data quality)

데이터 유형과 방법론에 대한 세부 사항

데이터 흐름(Data flow)

데이터 소스(Data Sources)

레드스톤(RedStone)은 분산형 데이터 소싱 프레임워크를 구현하여, 여러 공급자 카테고리에 걸친 전략적 중복성을 통해 운영 연속성을 보장하며, 일시적 중단과 잠재적 데이터 조작으로부터 보호한다.

소스 카테고리(Source Categories)

중앙화 거래소 통합(Centralized Exchange Integration)

바이낸스(Binance), 코인베이스(Coinbase), OKX, 바이비트(Bybit), 크라켄(Kraken) 등 업계 주요 거래소와 파생상품 플랫폼을 포함한 30개 이상의 프리미어 암호화폐 거래소와 직접 통합되어, 확립된 거래소 전반에서 포괄적인 가격 발견을 가능하게 한다.

탈중앙화 프로토콜 구현(Decentralized Protocol Implementation)

유니스왑(Uniswap), 커브 파이낸스(Curve Finance), 밸런서(Balancer), 아에로드롬(Aerodrome), 카멜롯(Camelot) 등 주요 자동화 마켓메이커(AMM)와 탈중앙화 거래소를 포함해 다중 블록체인 네트워크 전반의 50개 이상의 온체인 거래 프로토콜과 통합되어, 폭넓은 크로스체인 가격 검증을 보장한다.

애그리게이터 서비스 통합(Aggregator Service Integration)

카이코(Kaiko), 코인마켓캡(CoinMarketCap) 등 교차 출처 데이터 집계에 특화된 전문 시장 인텔리전스 제공자의 데이터를 통합하여, 직접 거래소 피드를 보완하는 시장 통계 지표를 제공한다.

기관 데이터 제공 네트워크(Institutional Data Provider Network)

상장지수펀드(ETF)와 토큰화된 전통 자산 등 정교한 금융 상품의 가격 데이터를 제공하는 전문 오프체인 기관들과 보안 API 연결을 유지하여, 신흥 디지털 자산 클래스에 대한 포괄적인 커버리지를 보장한다.

데이터 가져오기(Data Fetching)

데이터는 다수의 지리적 위치에 전략적으로 배치된 독립 기관과 단체가 유지하는 분산형 공급자 노드 네트워크를 통해 가져온다. 이 다중 운영자 아키텍처는 단일 장애 지점(SPoF), 운영자 담합 리스크, 지역 네트워크 장애를 완화하는 회복력 있는 인프라를 형성한다. 각 논리 노드는 고가용성(HA) 구성으로 운영되며, 마스터 인스턴스와 동기화된 새도 인스턴스로 구성되어 서비스 중단 시 원활히 운영을 이어간다.

암호학적 보증(Cryptographic Attestation)

각 공급자 노드는 모든 데이터 전송에 대해 강력한 암호학적 서명 프로토콜을 구현하여, 데이터 출처를 검증하고 운영자 신뢰성 지표에 대한 감사 가능한 기록을 유지한다.

품질 보증 프로토콜(Quality Assurance Protocol)

시스템은 다음과 같은 파라미터 전반에서 노드 성능을 평가하기 위한 종합적인 모니터링 메커니즘을 적용한다:

- 데이터 일관성과 운영 가동시간(operational uptime)의 지속적 모니터링
- 데이터 품질 및 전송 지연(latency)의 실시간 검증

경제적 보안 모델(Economic Security Model)

네트워크는 운영 우수성을 보장하기 위해 이중 인센티브 구조를 구현한다:

- 우수한 서비스 품질 유지를 위한 성과 기반 보상 분배
- 서비스 수준 협약(SLA) 위반 시 프로그램화된 슬래싱 조건과 함께 의무적 담보 예치

데이터 정제(Data Sanitization)

다음 검증 메커니즘은 서비스 신뢰성을 보장하기 위해 비정상적이거나 조작 가능성이 있는 가격 데이터를 탐지 및 필터링하는 종합적 프레임워크를 구성한다:

가격 변동 분석 기반 이상 탐지(Anomaly Detection via Price Movement Analysis)

자산군별 임계값을 활용해 기존 가격 경로에서 통계적으로 유의미한 편차를 보이는 값을 식별 및 제외하며, 상대적 측정 프레임워크를 사용한다.

시장 깊이 검증(Market Depth Validation)

DEX의 슬리피지 지표와 CEX의 시장 깊이 분석을 실시간으로 모니터링해, 유동성이 부족한 시장에서의 가격 데이터를 자동으로 필터링한다.

다중 소스 분산 분석(Multi-Source Variance Analysis)

통계 모델링을 적용하여 교차 출처 중앙값에서 큰 편차를 보이는 가격 입력을 식별 및 제외함으로써, 다중 가격 피드 전반에서 데이터 일관성과 신뢰성을 보장한다.

페그 준수 모니터링(Peg Compliance Monitoring)

스테이블코인의 USD, 리퀴드 스테이킹 파생상품의 ETH, 래핑된 비트코인 프로토콜의 BTC 등 자산별 기준 페그를 유지하며, 기준 페그에서 단기적으로 벗어난 값을 자동으로 필터링한다.

데이터 집계(Data Aggregation)

레드스톤은 자산 특성과 클라이언트 요구사항에 맞게 보정된 다중 집계 전략을 구현하여 최적의 가격 정확성과 신뢰성을 보장한다.

중앙값 기반 집계(Median-Based Aggregation)

정렬된 가격 입력 집합에서 중앙값을 선택하는 견고한 통계적 방법론으로, 극단적 이상값을 효과적으로 필터링한다. 이 방식은 가격 조작 시도에 본질적으로 강하며, 일부 출처에서 이상값을 보고하더라도 정확성을 유지한다.

시간가중평균가(TWAP, Time-Weighted Average Price)

30초~1시간의 설정 가능한 시간 창에서 가중 평균을 계산하는 고급 시계열 집계 메커니즘으로, 단기적 가격 변동성과 조작 시도를 효과적으로 완화한다. 다만, 급격한 시장 변화를 반영하는 데에는 일정 지연(latency)이 발생한다.

유동성가중평균가(LWAP, Liquidity-Weighted Average Price)

DEX의 슬리피지율, CEX의 호가창 깊이 등 시장 깊이 지표를 반영해 가중 평균을 계산하는 고급 유동성 민감형 방법론이다. 시스템은 각 시장의 가용 유동성에 따라 가격 가중치를 동적으로 조정하며, 더 높은 유동성과 거래 활동을 보이는 시장에 더 큰 가중치를 부여해 실행 가능한 시장 조건을 정확히 반영한다.

데이터 가용성(Data Availability)

레드스톤(RedStone)은 중복된 분산 경로를 갖춘 정교한 방송 인프라를 구현하여 최적의 데이터 가용성과 전송 신뢰성을 보장한다.

배포 채널(Distribution Channels)

시스템은 기업급, 지리적으로 분산된 게이트웨이 인프라와 고급 캐싱 기능을 결합하고, 가십 프로토콜(gossip protocol)을 활용하는 탈중앙화 노드 네트워크를 혼합한 하이브리드 배포 모델을 사용한다. 이 인프라는 서드파티 발행-구독 프로토콜인 Streamr와의 통합을 통해 강화되어, 회복력 있는 데이터 전파를 가능하게 한다.

릴레이 인프라(Relay Infrastructure)

시스템은 자동 가스비 최적화, 중복 RPC 엔드포인트와의 부하 분산 상호작용, 새도 인스턴스를 활용한고가용성 배포 등을 통해 원활한 트랜잭션 처리를 지원하는 릴레이 서비스 네트워크를 운영한다. 또한, Gelato와 같은 서드파티 트랜잭션 릴레이 서비스를 포함해, 폴백(fallback) 서비스로 추가적인 중복성을 확보한다.

데이터 검증(Data Verification)

최종 단계에서 레드스톤은 데이터 무결성과 적시성을 소비 지점에서 보장하기 위해 블록체인 고유의 보안 메커니즘을 활용하는 포괄적인 온체인 검증 프레임워크를 구현한다.

시간 검증(Temporal Validation)

시스템은 스마트 컨트랙트 계층 내에서 엄격한 타임스탬프 검증을 시행하며, 설정된 오래됨(staleness) 임계값을 초과하는 데이터 포인트는 자동으로 거부한다. 이를 통해 최신 시장 데이터만이 프로토콜 운영에 영향을 미치도록 보장한다.

운영자 권한 부여(Operator Authorization)

스마트 컨트랙트는 승인된 노드 운영자 레지스트리를 유지하며, 각 데이터 포인트를 등록된 출처의 암호학적 서명과 대조한다. 이를 통해 필수 보안 예치금과 서비스 수준을 유지하는 운영자의 데이터만 시스템에 반영된다.

합의 기반 가격 형성(Consensus-Based Price Formation)

프로토콜은 다수의 독립적인 가격 피드를 처리해 단일 기준 값을 생성하는 온체인 중앙값(medianization) 알고리즘을 구현한다:

- 유효 제출물의 최소 정족수(quorum) 요건
- 온체인 중앙값 계산을 통한 최종 가격 산정
- 이상 제출물에 대한 선택적·맞춤형 필터링

이 아키텍처는 가격 조작을 위해 다수의 독립적이고 담보화된 운영자가 협력해야만 가능하도록 설계되어, 시스템의 보안 임계치를 크게 높인다.

데이터 유형(Types of data)

오라클은 데이터 피드를 dApp 또는 블록체인에 직접 제공하여, 스마트 컨트랙트의 트랜잭션이 올바르게 실행되도록 한다. 지난 수년간 디파이 시장은 오라클 피드의 구성에 대해 다양한 기대와 요구를 보여왔다. 레드스톤은 해당 분야에서 개발자의 요구에 부응하는 혁신을 선도하고 있으며, 다음과 같은 데이터 피드 유형을 정의한다:

1. 시장 피드(Market Feed)

중앙화 및 탈중앙화 거래소에서 파생된 일반 시장 가격. 예: ETH / USD.

2. 계약 비율 피드(Contract Rate Feed)

환율(Exchange Rate) 또는 펀더멘털 피드(Fundamental Feed)라고도 불린다. 자산과 파생상품 간 비율로, 특정 프로토콜의 스마트 컨트랙트에서 도출된다. 예: Lido 컨트랙트에서 도출된 wstETH / stETH 비율, Ether.fi 컨트랙트에서 도출된 weETH / eETH 비율.

3. 순자산가치(NAV) 피드(Net Asset Value Feed)

기관 API(예: CESR 또는 SOFR) 또는 기타 소스로부터 가져온 데이터 피드로, 해당 데이터를 기반으로 발행된 토큰을 뒷받침하는 자산의 가치를 제공한다. 예: Blackrock BUIDL.

4. 준비금 증명 피드(Proof of Reserve Feed)

특정 오라클을 지원하기 위한 맞춤형 로직 구현. 예: 비트코인 스테이킹 오라클.

- 사례: Lombard의 비트코인 체인 시스템에서 발행된 LBTC 수량과 예치된 BTC 총량을 기준으로 도

출된 LBTC / BTC 가격 피드. 자세한 설명은 [여기](#) 참조.

피드 유형(Feed Type)	명목화(Denomination)	편차 원인 (Deviation Comes From)	전달 복잡성 (Complexity of Delivery)
시장 피드(Market Feed)	주로 USD, 다른 통화도 가능 (예: ETH / BTC)	시장 거래(Market trading)	상대적으로 단순 (Relatively simple)
계약 비율 피드 (Contract Rate Feed)	기초 자산 (예: 이더리움 LST의 경우 ETH)	스마트 컨트랙트의 특정 비율 변화	엠티 케이스 및 온체인 소스 이해 필요
NAV 피드(Net Asset Value Feed)	맞춤형, 예: CESR의 경우 ETH 스테이킹 수익률 %, 일반적으로 USD 값	맞춤형, 보통 API에서 24시간마다 업데이트	소스에 따라 다르며, 일반적으로 중간 난이도
준비금 증명 피드 (Proof of Reserve Feed)	맞춤형, 예: Lombard 내부 오라클의 경우 BTC	맞춤형, 예: Lombard의 경우 X분마다 업데이트	일반적으로 높은 작업 강도(Work-intensive)

롬바르드 LBTC 피드(Lombard LBTC feed)

아래 문서는 레드스톤(RedStone)이 LBTC 토큰의 온체인 근본 가치(fundamental value)를 어떻게 계산하고 전달하는지를 설명한다.

롬바르드 프로토콜과 LBTC(Lombard protocol and LBTC)

개요(Overview)

롬바르드 BTC 스테이킹 프로토콜(Lombard BTC Staking Protocol)은 사용자가 비트코인(BTC)을 신뢰할 수 있고 안전한 방식으로 스테이킹할 수 있게 하는 플랫폼이다. 이 과정은 비트코인의 보안을 지분증명(PoS) 시스템의 유연성과 통합하는 바빌론 네트워크(Babylon Network)에 의해 지원된다. 롬바르드 프로토콜은 비트코인 블록체인에서 1:1로 BTC로 담보된 유동 스테이킹 토큰(LST)인 LBTC를 도입했다. LBTC는 옴니체인 토큰으로 설계되어, 여러 EVM 체인에서 발행 및 전송될 수 있다. 현재 LBTC 발행은 이더리움 네트워크에서 가능하며, 롬바르드 팀은 지원 체인을 확장하기 위해 적극적으로 노력하고 있다.

LBTC 발행(Minting LBTC)

LBTC를 발행하기 위해 사용자는 먼저 BTC를 롬바르드 프로토콜이 제어하는 비트코인 블록체인의 특정 주소로 전송해 스테이킹한다. 이 단계에서 BTC는 잠금 상태가 되지만, 아직 LBTC는 발행되지 않는다. 필요한 블록체인 컨펌 (일반적으로 6회, 약 1시간 소요)이 완료되면, 사용자는 대상 체인(현재는 이더리움)과 발행할 LBTC 수량에 대한 암호학적 증명을 제출한다. 이후 LBTC가 선택된 EVM 체인에서 사용자의 지갑으로 직접 발행된다.

LBTC 소각(Burning LBTC)

LBTC를 소각하고 BTC를 언스테이킹하려면, 사용자는 EVM 체인에서 LBTC를 롬바르드 프로토콜로 전송해 소각 절차를 시작한다. 소각이 완료되면 비트코인 블록체인에 있는 BTC는 보안상의 이유로 7일간의 출금 대기 기간을 거친다. 이후 언스테이킹된 BTC가 해제되어 사용자가 지정한 비트코인 주소로 전송된다.

레드스톤이 LBTC/BTC 근본 가격을 전달하는 방식

개요(Overview)

레드스톤은 풀 모델과 푸시 모델을 모두 활용해 LBTC/BTC 근본 가격을 제공한다. 이 가격은 롬바르드 프로토콜이 보유한 BTC와 지원되는 모든 체인에서의 LBTC 토큰 총 공급량의 비율로 계산된다. 또한

total_unclaimed_lbtc(아직 발행되지 않았지만, 이미 해당 BTC가 롬바르드 프로토콜에 예치된 LBTC 토큰)와 **total_btc_unstakes_pending**(7일 출금 대기 중인 BTC 토큰) 값도 조정에 반영된다.

현재 값은 상한선이 1로 설정되어 있으며, 1이라는 값은 프로토콜의 건전성을 의미한다. 즉, 프로토콜이 보유한 BTC가 유통 중인 LBTC 토큰보다 많거나 같다는 것을 나타내며, 이는 완전 담보화 또는 초과 담보화를 보장한다. 예를 들어, 롬바르드가 제어하는 지갑에서 BTC의 절반이 사라지고, LBTC 공급량은 동일하게 유지된다면, 비율은 0.5로 떨어진다.

LBTC/BTC 근본 가격 계산 (풀 모델)

다음 알고리즘은 각 오라클 노드에서 독립적으로 20분마다 실행된다:

1. 롬바르드 API에서 BTC 주소 목록을 가져온다.
2. 각 새로운 주소(레드스톤이 이전에 처리하지 않은 경우)에 대해:
 1. 롬바르드의 루트 공개키를 사용해 주소 소유권을 검증한다.
 2. 주소를 오라클 노드의 영구 저장소에 저장한다.
3. 모든 저장된 주소의 비트코인 잔액을 합산해 **total_btc_balance**를 계산한다.
4. 롬바르드 API를 사용해 **total_btc_unstakes_pending**을 계산한다. (음수일 경우 0으로 처리)
5. 지원되는 모든 EVM 체인에서 LBTC 토큰의 총 공급량을 합산해 **total_supply_on_evms**를 계산한다.
6. 롬바르드 API를 사용해 **total_unclaimed_lbtc**를 계산한다. (음수일 경우 0으로 처리)
7. 최종 비율을 다음 공식으로 계산한다:
$$(\text{total_btc_balance} - \text{total_btc_unstakes_pending}) / (\text{total_supply_on_evms} + \text{total_unclaimed_lbtc})$$
8. 해당 값을 타임스탬프와 함께 오라클 노드의 영구 저장소에 저장한다.
9. 최근 6시간 동안 저장된 값의 중앙값을 계산한다. 이 단계는 체인 간 지연으로 인한 일시적 급등을 무시하고 견고성과 보안을 강화하는 데 중요하다.
10. 값을 개인 키로 서명하고, 레드스톤 분산 데이터 계층(DDL)에 방송한다.

이더리움에서 LBTC 근본 가격 저장 (푸시 모델)

풀 모델에서 산출된 값은 이더리움 네트워크에 저장된다. 이는 표준 AggregatorV3Interface 인터페이스를 통해 다음 컨트랙트에서 확인할 수 있다:

[0xb415eAA355D8440ac7eCB602D3fb67ccC1f0bc81](#)

이 값은 최소 24시간마다 업데이트되며, 온체인 값이 현재 값과 1% 이상 차이가 발생하면 더 자주 갱신된다.

- [레드스톤 앱에서 피드 상세 보기](#)

보안(Security)

레드스톤 보안 관행(RedStone security practices)

보안 주도 설계(Security driven design)

레드스톤 아키텍처는 표준화된 인터페이스로 상호 연결된 모듈형 구성요소를 사용하는 분산 시스템 설계 패턴을 채택하여, 전략적 중복성과 구성요소 격리를 통해 단일 장애 지점(SPoF)을 제거한다. 이 접근법은 의료기기와 항공 시스템 같은 중요 분야에서 적용되는 고신뢰성 공학(high-reliability engineering) 관행을 반영한다. 항공기가 다중 독립 비행 제어 컴퓨터를 사용하고 의료기기가 중복 모니터링 시스템을 사용하는 것처럼, 레드스톤의 아키텍처도 구성요소 독립성과 페일세이프(fail-safe) 설계 원칙을 통해 시스템 무결성을 보장한다.

데이터 생산과 소비의 분리(Decoupling data production and consumption)

시스템은 검증 가능한 보증 체인(attestation chain)을 갖춘 암호학적 서명이 적용된 데이터 단위들을 사용하여, 데이터 출처(provenance)와 진위(authenticity)를 독립적으로 검증할 수 있게 한다. 이러한 아키텍처는 데이터 생산, 전송, 소비 계층 간의 논리적 분리를 형성하여, 각 구성요소가 독립적으로 운영되고 서로 다른 주체에 의해 중복 구현될 수 있도록 한다.

중복 데이터 소스(Redundant data sources)

데이터 생산의 신뢰성은 소스 다변화와 중복 처리 파이프라인을 통해 달성된다. 시스템은 직접 거래소 피드와 애그리게이터 서비스를 함께 통합하고, “데이터 집계(Data Aggregation)” 섹션에 상세한 복수의 동시 집계 방법론을 적용한다. 이 다중 방법론 접근법은 단일 데이터 처리 알고리즘의 잠재적 취약점이 시스템 무결성을 훼손하지 못하도록 보장한다.

분산 노드 보안(Distributed Node Security)

분산 노드 네트워크 아키텍처는 운영의 탈중앙화를 통해 개인 키 유출이나 경제적 강압 시도에 따른 위험을 완화한다. 지리적으로 분산된 노드 배치는 지역 네트워크 장애나 데이터센터 정전 등 인프라 레벨 장애에 대한 추가적인 회복력을 제공한다.

시스템은 특화 최적화를 갖춘 이질적(heterogeneous) 노드 구현을 유지한다. 예컨대 다수의 온체인 데이터 소스를 처리하는 고처리량(high-throughput) 노드와, 중앙화 거래소 데이터를 위한 저지연(low-latency) 노드가 있다. 이러한 구현은 표준화된 데이터 교환 프로토콜을 준수하여, 특정 기술 스택이나 아키텍처 패턴에 특유한 취약점으로부터 보호받으면서도 상호운용성을 보장한다.

데이터 가용성(Data availability)

데이터 전송의 신뢰성은 기업급 웹 게이트웨이, 경량 오픈소스 P2P 노드, MQTT 메시지 브로커, Streamr 네트워크 통합으로 구성된 4중 중복 분배 인프라를 통해 달성된다. 이 아키텍처는 프로토콜 수준의 장애 조치(failover) 메커니즘으로 지속적인 데이터 가용성을 보장한다.

장애 조치 시스템 아키텍처(Failover System Architecture)

온체인 데이터 제출은 운영 연속성을 유지하기 위한 다중 장애 조치 시스템을 구현한다. 1차 릴레이 네트워크는 성능 저하가 감지되면 자동으로 활성화되는 섀도(shadow) 네트워크와 함께 동작한다. 추가적인 회복력은 Gelato와 같은 서드파티 릴레이 서비스 통합을 통해 제공된다. 시스템의 신뢰 불필요(trustless) 아키텍처는 공개 데이터 제출 프로토콜을 활용한 외부 주체의 무허가(permissionless) 운영을 가능하게 하여, 궁극적 장애 조치 수단으로 기능한다.

개발 프로세스(Development process)

레드스톤은 엄격한 프로세스 통제와 다층 검증 절차를 통해 시스템 무결성을 우선하는 포괄적인 보안 주도 개발 생명 주기(SDLC)를 구현한다. 개발 워크플로는 설계 문서부터 시작하며, 구현 전 잠재적 보안 영향을 철저히 평가하기 위해 기능 간 동료 검토(peer review)를 거친다.

코드 개발 단계에서는 다층 검토 정책을 집행한다. 변경 사항은 기술적 구현에 중점을 둔 1차 리뷰, 잠재적 공격 벡터와 엡지 케이스를 점검하는 보안 리뷰를 순차적으로 통과해야 한다. 보안상 중요한 구성요소는 외부 보안 전문가의 추가 검토를 요구하여, 보안 핵심 구현에 대한 독립적 검증을 확보한다.

지속적 통합(CI) 파이프라인은 정적 코드 분석, 의존성 취약점 스캐닝, 스마트 컨트랙트 특화 보안 도구 등 자동화된 보안 통제를 적용한다. 이러한 자동 점검은 난수 입력 기반의 성질(property) 검증 테스트와, 구성요소 간 보안 속성을 확인하는 통합 테스트로 보완된다. 테스트 프레임워크에는 다양한 공격 시나리오를 시뮬레이션하고 시스템의 복원력을 검증하는 전용 보안 테스트 스위트가 포함된다.

배포 절차는 각 단계에서의 보안 검증을 강조하는 블루-그린(blue-green) 방식으로 진행된다. 신규 배포는 먼저 병렬 인프라 스택에 릴리스되어 침투 테스트와 자동 보안 스캐닝을 포함한 포괄적 보안 검증을 수행한다. 배포 프로세스에는 보안 관련 이상 징후가 감지될 경우 자동 롤백 트리거가 포함되어 있다. 운영 전환은 모든 보안 검증 단계가 성공적으로 완료되고 보안 팀의 수동 승인이 이루어진 후에만 진행된다.

개발 문화는 정기 교육, 인시던트 대응 훈련, 블록체인 생태계 전반의 보안 사건 사후 분석(post-mortem)을 통해 보안 인식을 강화한다. 팀 구성원들은 기능 설계의 초기 단계부터 잠재적 공격 벡터와 오남용 시나리오를 고려하는 보안 우선 관점(security-first lens)으로 의사결정을 내리도록 장려된다. 이러한 문화적 초점은 선제적 보안 기여에 대한 인정과, 잠재적 취약점을 발굴·해결하기 위한 정기 보안 중심 브레인스토밍 세션을 통해 더욱 강화된다.

예방 및 모니터링(Prevention & monitoring)

레드스톤(RedStone)은 시스템 무결성과 보안을 보장하기 위해 다층 검증에 걸친 포괄적인 테스트 전략을 구현한다.

- 단위 테스트(Unit test)는 개별 구성요소의 세부적인 검증을 수행하며, 특히 보안 핵심 기능의 엡지 케이스

와 오류 처리에 중점을 둔다.

- 통합 테스트(Integration test)는 시스템 구성요소 간 상호작용을 검증하며, 암호학적 연산과 데이터 흐름 무결성을 포함한다.
- 종단 간 테스트(End-to-end test)는 전체 인프라 스택 전반의 운영 시나리오를 시뮬레이션한다.
- 특수 크래시 테스트(Crash testing)는 시스템의 다양한 수준에서 인위적으로 장애를 발생시켜, 점진적 성능 저하와 복구 메커니즘을 검증한다.

모니터링 인프라는 분산 네트워크로 운영되며, 여러 차원의 시스템 상태를 추적한다.

- 성능 모니터링은 지연(latency)과 처리량(throughput) 지표를 자산별 임계값과 함께 추적한다.
- 데이터 품질 모니터링은 가격 일관성과 시장 깊이 파라미터를 검증한다.
- 인프라 모니터링은 분산 시스템 구성요소 전반의 노드 상태, 네트워크 연결성, 리소스 사용량을 추적한다.
- 보안 모니터링은 잠재적 공격 패턴 감지에 초점을 맞추며, 비정상 데이터 제출, 의심스러운 스마트 컨트랙트 상호작용, 조작 시도 등을 탐지한다.

각 모니터링 모듈은 독립적인 데이터 영속성과 중복 저장소를 유지하여, 사건 발생 후(post-incident) 분석을 위한 이력 데이터 가용성을 보장한다.

알림 관리 시스템은 명확히 정의된 에스컬레이션 정책을 갖춘 계층형 대응 프레임워크를 구현한다.

- 중요한 알림은 중복 커뮤니케이션 채널을 통해 당직자에게 즉시 전달되며, 지정된 시간 내 확인이 없으면 자동 에스컬레이션된다.
- 24/7 당직 체계를 유지하며, 각 교대에는 1차 및 백업 담당자가 지정되고, 특정 사건 유형에 따라 도메인 전문가가 활성화될 수 있다.
- 에스컬레이션 정책은 고위 기술 직원 또는 관리 인력의 개입 기준과 응답 시간 프레임에 명확히 정의한다.
- 각 알림 카테고리에는 초기 대응 절차와 진단 단계를 정의한 플레이북이 있어, 대응 인력에 관계없이 일관된 사건 처리가 가능하다.

시스템은 모든 알림, 대응, 해결 단계를 포괄적으로 감사 기록(audit trail)으로 유지하여, 모니터링 임계값과 대응 절차의 지속적인 개선을 가능하게 한다. 전체 모니터링 인프라는 중복 기술 스택과 풀백 서비스로 구현되어, 최대한의 신뢰성과 복원력을 보장한다.

보안 감사(Audits)

레드스톤은 보안 보증 프레임워크의 핵심 요소로서 엄격한 외부 보안 감사 프로그램을 유지한다.

여러 독립적인 블록체인 인프라 전문 보안 회사가 프로덕션 배포 이전에 핵심 프로토콜 구성요소와 주요 시스템 변경 사항에 대한 포괄적인 감사를 수행한다.

감사 범위는 스마트 컨트랙트, 암호학적 보증 메커니즘, 릴레이 네트워크를 포함하는 전체 인프라 스택을 아우른다. 중요 발견 사항은 즉시 수정되며, 이후 검증 절차를 거친다. 모든 감사 보고서는 아래와 같이 공개된다:

구성요소(Component)	상태(Status)	감사 기관(Auditor)	보고서(Report)
Oracle Phase II	Completed	ABDK	ABDKRedStone_OraclePhaseII_v_2_o_1.pdf
Token And Vesting	Completed	ABDK	ABDK_RedStone_TokenAndVesting_v_3_o.pdf
EVM connector	Completed	ABDK	Audit_ABDK_EVM connector.pdf
Adapter	Completed	ABDK	Audit_ABDK_Adapter.pdf
Merged adapter	Completed	ABDK	Audit_ABDK_Merged adapter.pdf
Multifeed contracts	Completed	ABDK	Audit_ABDK_Multifeed contracts.pdf
Oracles 2.0	Completed	ABDK	Audit_ABDK_Oracles v2.pdf
Adapter contracts	Completed	AuditOne	Audit_Auditone_Adapter contracts.pdf
Fuel connector	Completed	Halborn	Audit_Halborn_Fuel(Sway)_Sep.pdf
TGE contract	Completed	Codespect	Audit_Codespect_TGE.pdf
TGE contract	Completed	Halborn	Audit_Halborn_TGE.pdf
Oracles	Completed	Cantina (SpearBit)	Audit_Cantina(SpearBit)_Oracles.pdf
Sui connector	Completed	BitsLab	Audit_MoveBit (BitsLab)_Sui connector.pdf
Aggregator	Completed	ABDK	Audit_ABDK_Spark.pdf
Aggregator	Completed	Cantina (SpearBit)	Audit_Cantina(Spearbit)_Spark.pdf

데이터 제공자(Data Providers)

소개(Introduction)

레드스톤(RedStone)은 시장에서 선도적 블록체인 오라클 제공자 중 하나이지만, 큰 힘에는 큰 책임이 따른다. 가격 조작의 (이미 낮은) 위험을 최소화하기 위해, 레드스톤은 다른 검증되고 신뢰할 수 있는 기업들이 오라클 데이터를 게이트웨이에 전송할 수 있도록 허용한다. 이를 통해 안전성과 투명성을 높이고 데이터 품질을 개선한다.

이러한 회사를 **외부 오라클 제공자(External oracle providers, EOP)**라고 부른다.

외부 오라클 제공자가 되기(Becoming External oracle provider)

모든 회사가 외부 오라클 제공자(EOP)가 될 수 있는 것은 아니다. 기관은 법적, 기술적 조건을 충족해야 하며, 전체 과정은 3단계 — 법적 분석, 스테이징 참여, 프로덕션 참여 — 로 나눌 수 있다.

법적 분석(Legal Analysis)

이 단계가 끝나면 EOP는 환율 조작 같은 악의적 행위에 따른 법적 결과를 이해하고, 보상 메커니즘이 어떻게 작동하는지도 알게 된다.

스테이징(Staging)

법적 문제가 모두 해결되면, EOP는 자체 오라클 노드를 설정하기 시작한다. 이 단계에서 레드스톤과 EOP의 데브옵스 팀이 연락해 모든 설정을 진행한다.

EOP는 스테이징 환경에 먼저 연결되고, 레드스톤은 배포 안정성을 면밀히 모니터링한다. 다운타임뿐만 아니라 제공된 가격도 기록하며, 프로덕션으로 이동하기 전에 모든 문제를 파악하고 해결한다.

이 단계는 필요한 만큼 지속되며, 최소 30일 이상이다. 레드스톤은 EOP 팀과 지속적으로 연락하며 문제를 즉시 해결할 수 있도록 돕는다.

프로덕션(Production)

EOP가 스테이징에서 안정적으로 운영되고 양측이 동의하면, 프로덕션으로 이동한다. 이후 레드스톤은 계속해서 EOP를 모니터링하며, 이제 데이터는 프로덕션 게이트웨이에서 사용 가능하다.

info 데이터 가용성(Data availability)

레드스톤 소비자는 귀하의 노드가 게시하는 데이터를 모든 지원 체인에서 사용할 수 있다.

레드스톤 오라클 아키텍처에 대한 자세한 내용은 [여기](#)에서 확인할 수 있다.

배포(Deploy)

이 섹션에서는 Docker Compose를 사용해 레드스톤 노드를 실행하는 방법을 설명한다.

caution

프로덕션 배포의 경우 Kubernetes와 같은 더 정교한 도구 사용을 고려해야 한다.

요구사항(Requirements)

- 안정적인 인터넷 연결
- 2 CPU, 2GB RAM
- 최소 30GB 스토리지 (주로 로그용)
- [Docker](#) 및 [Docker Compose](#)

tip

레드스톤 노드 실행을 위한 하드웨어 요구사항은 매우 낮지만, 충분한 마진을 확보하는 것이 좋다. 리소스를 여유 있게 할당할수록 성능이 향상되고 문제가 발생할 가능성이 줄어든다.

Docker Compose 예시(Docker Compose Example)

아래는 로컬에서 레드스톤 오라클 노드를 실행하기 위해 사용할 수 있는 간단한 docker-compose.yml 예시다. 이 구성을 복사해 필요에 맞게 조정하면 된다.

```
services:
  redstone-oracle-node:
    image: public.ecr.aws/y7v2w8b2/redstone-oracle-
node@sha256:07eb1cc4aa3a4f0275c2ef5c2f9a95af06150e35211e20f66f9b24ab1c05cef7
    restart: always
    depends_on:
      - redstone-kms
    networks:
      - public_network
      - private_network
    volumes:
      - redstone-oracle-node:/oracle-node-level-db
    environment:
      OVERRIDE_DIRECT_CACHE_SERVICE_URLS: '["https://httpbin.org/anything"]'
      OVERRIDE_MANIFEST_USING_FILE: ./manifests/dev/dev.json
      PRICE_CACHE_LOCATION: /oracle-node-level-db
      REMOTE_SIGNER_URL: http://redstone-kms:4499
      ENABLE_REMOTE_SIGNER: true

  redstone-kms:
    restart: always
    image:
public.ecr.aws/y7v2w8b2/kms@sha256:d069bc9afcd1b4e6884e2d4e530d90c94db0aaf1a2
265d7facb4f4e2d2fefb3d
    networks:
      - private_network
    expose:
      - "4499"
    environment:
      KMS_PRIVATE_KEY_FILE: /run/secrets/private_key
    secrets:
      - private_key

secrets:
  private_key:
    file: private_key.txt

volumes:
  redstone-oracle-node:

networks:
  public_network:
    driver: bridge
  private_network:
```

internal: true # 이 설정은 네트워크를 프라이빗으로 보장한다

서비스 구성(Services)

RedStone KMS (Key Management Service)

RedStone KMS의 유일한 목적은 개인 키 관련 모든 작업을 안전하게 처리하는 것이다.
이 서비스는 오라클 노드가 가져온 데이터를 서명하고 EVM 주소를 반환한다.
이 서비스만이 개인 키에 접근할 수 있어야 하며, [RedStone KMS Docker](#) 이미지를 사용해야 한다.

매개변수(Param)	설명(Description)	예시 값(Example value)
KMS_PRIVATE_KEY_FILE	개인 키가 포함된 파일의 경로. 파일에는 개인 키 한 줄만 포함되어야 한다.	KMS_PRIVATE_KEY_FILE=/run/secrets/private_key
KMS_PRIVATE_KEY	16진수(HEX) 인코딩된 키. 키를 직접 전달하는 대체 방법.	KMS_PRIVATE_KEY=0xYOUR_PRIVATE_KEY
KMS_ADDRESS	서비스 바인딩 주소.	KMS_ADDRESS=0.0.0:4499

RedStone Node

RedStone 오라클 노드는 주요 퍼블릭 API에서 데이터를 가져와 개인 키로 서명하고, 이를 Streamr 네트워크 및 RedStone 데이터 배포 계층(DDL)에 브로드캐스트한다.
환경 변수를 통해 설정할 수 있다.

매개변수(Param)	설명(Description)	예시 값(Example value)
ENABLE_REMOTE_SIGNER	서명을 원격 서명자(Remote signer)에 위임한다. 이 이미지에서만 개인 ECDSA 키에 접근 가능하다.	ENABLE_REMOTE_SIGNER=true
REMOTE_SIGNER_URL	RedStone의 서명자가 요청을 수신하는 URL. 예를 들어 Kubernetes에서는 오라클 노드와 서명자를 같은 POD 내에서 실행하는 것을 권장한다. 기본값은 http://localhost:4499.	REMOTE_SIGNER_URL=http://localhost:4499
OVERRIDE_DIRECT_CACHE_SERVICE_URLS	RedStone 데이터 배포 계층(DDL)의 개인 게이트웨이 URL. 로컬 노드를 실행할 경우 ["https://httpbin.org/anything"] 을 지정하면 된다. 그러나 프로덕션 환경에서는 RedStone 팀에 공식 요청을 해야 한다.	OVERRIDE_DIRECT_CACHE_SERVICE_URLS=["https://xxx.yyy.secret-url-1.com", "https://zzz.aaa.secret-url-2.com"]
OVERRIDE_MANIFEST_USING_FILE	매니페스트 파일의 경로. 매니페스트는 공급자가 제공하는 데이터에 대한 의무사항을 정의하는 공개 JSON 파일로, 수집 주기, 토큰, 데이터 소스 및 기타 기술적 세부 정보를 설정한다. 사용 가능한 매니페스트는 여기 에서 확인할 수 있다.	OVERRIDE_MANIFEST_USING_FILE=./manifests/dev/dev.json
PRICE_CACHE_LOCATION	로컬 가격 캐시의 경로. 각 RedStone 오라클 노드는 파일 기반의 간단한 캐시 시스템을 사용하며, 최근 15분 내에 가져온 값을 저장한다. 이 캐시는 값의 변동폭 점검, 이상치 필터링, 가격 조작 공격 방지를 위해 사용된다. Level DB 인스턴스를 수동으로 생성할 필요는 없으며, 노드 최초 실행 시 지정된 경로에 자동 생성된다.	PRICE_CACHE_LOCATION=/oracle-node-level-db
ENABLE_REMOTE_SIGNER (중복 표기)	RedStone KMS를 통한 서명 기능 활성화.	ENABLE_REMOTE_SIGNER=true

커스텀 로컬 매니페스트(Custom local manifest)

Docker를 사용해 오라클 노드를 실행하면서 사용자 정의 매니페스트를 적용하려면, 로컬 시스템의 매니페스트 파일을 바인드 마운트(bind mount)로 컨테이너에 연결하고 `OVERRIDE_MANIFEST_USING_FILE` 환경 변수를 해당 경로로 업데이트해야 한다.

모범 사례(Best practices)

장애 조치 기능(Failover Capabilities)

항상 두 개 이상의 레드스톤(RedStone) 노드를 동시에 운영하는 것이 권장된다. 이 중 하나는 Streamr 게시 기능을 비활성화할 수 있다(즉, `OVERRIDE_DIRECT_CACHE_SERVICE_URLS` 환경 변수에 Streamr 게이트웨이 URL이 포함되지 않음).

능동형 모니터링(Active monitoring)

문제가 발생하기 전에 이를 예측하고 해결하기 위해서는 **사전 예방적(proactive) 모니터링 접근 방식**을 구현해야 한다. 모니터링의 주요 초점은 다음과 같다:

- 노드 로그(Node Logs)
- CPU 및 메모리 사용량(CPU and Memory usage)
- 자체 모니터링 서비스를 구축하여 노드의 데이터가 소비자에게 정상적으로 제공되고 자주 업데이트되는지 확인

개인 키 보안(Secure your private keys)

개인 키를 안전하게 저장하기 위해 최선의 보안 관행을 반드시 적용해야 한다. 만약 키 중 하나라도 유출되었다고 의심되는 경우, 즉시 레드스톤 팀에 알리고 조치를 취해야 한다.

리스테이킹 운영자(Restaking operators, AVS)

리스테이킹(Restaking)

리스테이킹은 지분증명(PoS) 시스템에서 이미 스테이킹된 토큰을 재활용하여 추가 수익을 창출할 수 있도록 하는 메커니즘이다. 이를 통해 사용자는 동일한 토큰을 여러 스테이킹 기회에 동시에 참여시켜 효용을 극대화할 수 있다. 이는 더 높은 수익을 얻을 수 있는 가능성을 제공하지만, 동시에 추가 슬래싱(slaashing) 위험을 수반한다.

슬래싱이란, 검증자가 다운타임 또는 이중 서명(double-signing)과 같은 프로토콜 위반을 할 경우 지분 일부 또는 전부가 벌금으로 차감되는 과정을 말한다. 따라서 리스테이킹은 잠재적 수익을 높일 수 있지만, 리스테이킹 대상 프로토콜에 문제가 발생할 경우 자산 손실 위험도 커진다.

리스트레이킹 메커니즘은 새로운 프로토콜이 보안을 확보하는 데 특히 유용하다. 새로운 토큰을 만들고 스테이커를 모집하는 대신, 기존 스테이킹 인프라를 활용할 수 있기 때문이다. 리스테이킹을 지원하는 플랫폼을 이용하면 신규 프로토콜은 이미 확립된 네트워크의 스테이킹 토큰 보안을 활용할 수 있다.

이 모델은 신규 프로젝트의 진입 장벽을 낮춰, 보안 인프라 구축보다는 혁신과 개발에 집중할 수 있도록 돕는다. 이 접근법은 즉각적인 보안을 제공할 뿐 아니라, 스테이커의 인센티브를 여러 네트워크의 성공과 연계시켜 더 상호 연결되고 안전한 블록체인 생태계를 조성한다.

EigenLayer는 리스테이킹을 가능하게 하는 대표적인 플랫폼이다. 이 프레임워크에서 추가 보안을 필요로 하는 프로토콜은 **능동 검증 서비스(Actively Validated Services, AVS)** 라고 한다. 리스테이커는 AVS가 제공하는 서비스를 직접 검증하는 노드를 운영하거나, 자신의 스테이킹 토큰을 기존 운영자에게 위임(delegate)하여 검증 작업을 맡길 수 있다.

이중 선택 구조 덕분에 리스테이커는 본인의 참여 수준과 위험도를 선택할 수 있다 — 직접 검증을 수행해 보안을 강화하거나, 혹은 위임을 통해 보다 수동적으로 보상을 얻는 방식이다.

EigenLayer 모델은 기존 스테이킹 풀을 보안이 필요한 신규 프로젝트와 연결함으로써, 더 안전하고 유연한 블록체인 환경을 구축할 수 있는 리스테이킹의 잠재력을 보여준다.

오라클 분야에서의 AVS 사례로는 RedStone AVS가 있다. 이 모델을 활용해 안정적이고 신뢰할 수 있는 오라클 서비스를 제공한다.

레드스톤 AVS(RedStone AVS)

레드스톤 AVS는 리스테이킹(restaking) 개념을 활용하여 블록체인으로 가격 데이터를 전달하는 탈중앙화 프로세스를 구현한 혁신적인 솔루션이다. 이 시스템은 Othentic 프레임워크 위에서 구축되어, 가격 피드의 신뢰성과 보안을 모두 보장한다.

프로세스는 다수의 오라클 노드로부터 가격 데이터를 수집하는 것에서 시작된다. 수집된 데이터 포인트는 정확성, 일관성, 디지털 서명의 유효성 등 여러 단계의 검증을 거친다. 이 초기 검증을 통과한 데이터로부터 중앙값(median) 가격과 해당 타임스탬프가 계산된다. 이렇게 계산된 결과는 이후 AVS 운영자(operator)에 의해 검증된다. 운영자가 결과의 정확성을 확인하면, 해당 데이터가 블록체인으로 전송된다. 이 과정은 블록체인에 전달되는 가격 데이터가 정확하고 신뢰할 수 있음을 보장하며, 탈중앙화 금융(DeFi) 애플리케이션에 높은 수준의 신뢰성과 보안을 제공한다.

AVS 기반 푸시 모델 구현(Implementation of the Push Model with AVS)

레드스톤 AVS 솔루션은 블록체인으로 데이터를 전달하는 푸시(Push) 모델의 새로운 구현 방식을 제시한다. 기존 푸시 모델에서는 오프체인 릴레이어(relayer)가 데이터를 수집하여 블록체인으로 전송하고, 그 후 온체인에서 검증 과정을 수행했다. 이 방식은 효과적이지만, 온체인에서 검증 절차를 모두 수행해야 하므로 가스 사용량이 많아지는 단점이 있었다.

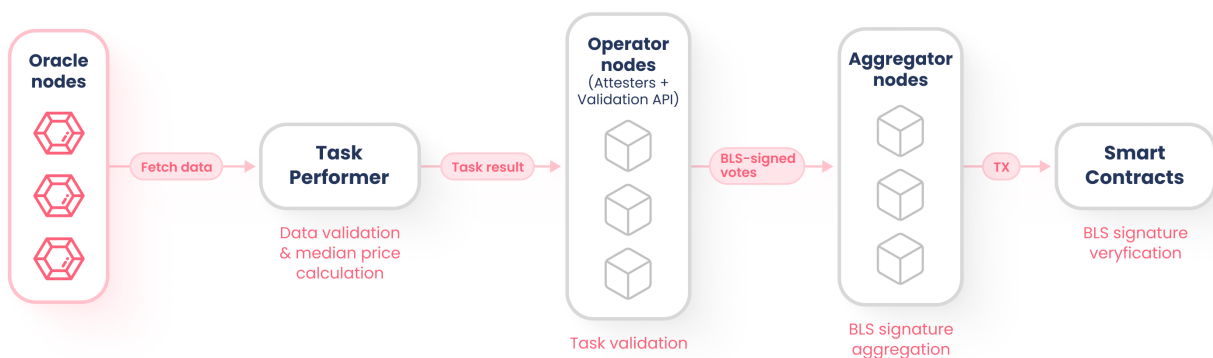
이에 반해 AVS 기반의 새로운 구현 방식은 검증 과정을 오프체인에서 수행함으로써 온체인 가스 소비를 크게 줄인다. 이 모델에서는 오라클 노드에서 수집된 가격 데이터가 오프체인 환경에서 엄격한 검증을 거쳐 정확성과 일관성을 확보한다. 이 검증 과정에는 디지털 서명 확인을 통한 데이터 진위 검증도 포함된다. 오프체인 검증이 완료되면, 중앙값 가격과 타임스탬프 결과가 AVS 운영자에 의해 다시 검증된다. 운영자는 오프체인 계산 결과가 변조되지 않았음을 확인하고, 검증된 결과와 정확성에 대한 서명 확인 데이터를 함께 블록체인으로 전송한다.

온체인에서는 AVS 운영자들의 집계된 BLS 서명만을 검증한다. 이 단순화된 프로세스는 데이터 수집 규모에 관계없이 높은 확장성을 제공한다. 전통적인 방식에서는 릴레이어가 중앙값에 가까운 일부 가격만 선택해 블록체인에 전송해야 했다. 하지만 AVS 접근법에서는 최종 중앙값 가격과 타임스탬프만 블록체인에 제출되므로, 가스 사용량은 참여한 오라클 노드 수에 따라 증가하지 않는다.

이 접근법은 온체인 운영 비용을 크게 절감할 뿐만 아니라, 효율성을 유지하면서 훨씬 더 많은 데이터 입력을 처리할 수 있는 확장성을 제공한다.

서비스 구성요소(Service Components)

레드스톤 AVS는 여러 핵심 구성요소로 구성되어 있으며, 각 구성요소는 검증 및 집계 과정에서 중요한 역할을 수행한다.



태스크 퍼포머(Task Performer)

태스크 퍼포머는 DDL(Data Distribution Layer)을 통해 여러 오라클 노드에서 데이터를 가져온다. 가져온 데이터의 정확성, 일관성, 그리고 서명(Signature)의 적합성을 검증한 후, 각 노드에서 수집한 데이터로부터 타임스탬프를 추출하고 중앙값(median) 가격을 계산한다. 또한 스마트 컨트랙트에서 반환된 현재 가격과 타임스탬프를 확인한다. 가격이 기준 편차를 초과하거나 마지막 업데이트 이후 일정 시간이 지나면, 계산된 결과를 집계기(Aggregator)에 전송한다.

어테스터(Attester)

어테스터는 태스크 퍼포머가 계산한 결과를 검증하고, 그 결과에 찬성하거나 반대하는 투표를 한다. 각 투표는 반드시 어테스터의 서명으로 확인되어야 한다. 검증 과정은 검증 API(Validation API)를 통해 수행된다.

검증 API(Validation API)

검증 API는 태스크 퍼포머가 계산한 결과의 정확성을 검증하는 서비스이다. 이 API는 오라클 노드에서 반환된 데이터를 확인하며, 각 노드의 공개키(public key)가 사전에 정의된 화이트리스트에 포함되어 있는지 검사한다. 또한 퍼포머가 계산한 중앙값과 타임스탬프의 정확성을 검증하고, 그 결과를 불리언(Boolean) 값으로 어테스터에게 반환한다.

집계기(Aggregator)

집계기는 태스크 퍼포머가 계산한 결과를 수신하는 허브 역할을 한다. 퍼포머의 결과는 미리 AVS 네트워크에 등록된 어테스터 풀(pool)로 전송된다. 집계기는 어테스터로부터 발생하는 이벤트, 즉 검증된 결과에 대한 서명된 찬반투표를 모니터링한다. 이러한 투표들은 집계되어 단일 BLS 서명으로 결합된다. 마지막으로, 태스크 퍼포머의 통합된 결과와 BLS 서명이 함께 블록체인에 배포된 스마트 컨트랙트로 전송된다.

스마트 컨트랙트(Smart contracts)

어테스터에 의해 검증된 후, 태스크 퍼포머의 결과는 블록체인으로 전송된다. 스마트 컨트랙트는 수신된 데이터를 재검증할 필요가 없으며, 해당 데이터가 충분한 수의 어테스터로부터 서명되었는지만 확인하면 된다. 이 방식은 가스비를 크게 절감한다.

새로운 가격이 충분한 투표를 받으면, 스마트 컨트랙트에 해당 가격과 새로운 타임스탬프가 함께 저장된다. 또한 어테스터는 자신들의 검증 활동에 대해 컨트랙트 내에서 보상을 받으며, 이 보상은 나중에 청구할 수 있다.

info

계산 결과가 전송되는 *AttestationCenter* 컨트랙트 주소는 다음과 같다.

- **Testnet:** [0x6af9B9272fc72CaC55ccDF6c2BC2c5703a65a187](#)
- **Mainnet:** [0x2B766957ce3dbab9eC4b227f5086855CeE7a1ad6](#)

네트워크 참여 및 역할(Network Participation and Roles)

레드스톤은 네트워크 초기 단계에서 태스크 퍼포머와 집계기를 직접 운영한다. 하지만, 외부 운영자들이 네트워크의 탈중앙화에 기여하도록 적극적으로 참여를 권장한다.

참여하려면, 운영자는 어테스터(Attester)와 해당 검증 API(Validation API)를 설정해야 한다. 운영자 구성, 등록, 실행 방법에 대한 자세한 내용은 다음 가이드를 참조하라: [운영자 실행 가이드\(running-avs-operator-mainnet\)](#)

보상(Rewards)

레드스톤 AVS(RedStone AVS)는 데이터의 탈중앙화 검증에 참여하는 운영자(operator)와 스테이커(staker)에게 보상을 제공한다. 이 섹션에서는 투표권(voting power), 보상 계산 및 분배 메커니즘, 그리고 슬래싱(slashing) 상태에 대해 설명한다.

info

레드스톤 AVS의 모든 스테이커는 **EIGEN 토큰으로 25% 추가 보너스**를 받는다. 이는 기존 RED 보상 위에 추가로 지급되며, 주기적으로 분배된다. 공식 발표는 X(트위터) 계정 [@redstone_defi](#)에서 확인할 수 있다.

투표권(Voting Power)

투표권은 레드스톤 AVS의 핵심 개념으로, 검증 작업(task validation) 과정에서 운영자의 영향력을 나타낸다. 특정 자산의 가격이 포함된 태스크가 제출될 때마다 여러 운영자가 이를 독립적으로 검증한다. 태스크가 스마트 컨트랙트에 반영되기 위해서는, 전체 투표권의 **2/3 이상을 가진 운영자들의 찬성 검증(positive validation)**을 받아야 한다.

운영자의 투표권은 AVS에서 지원하는 스테이킹 계약(Strategy) 내에서 보유한 지분에 따라 결정된다. 현재 레드스톤 AVS는 다음 세 가지 스테이킹 전략을 지원한다:

1. RED Strategy

(스테이킹 계약: [0x903a1FF023a35EfeD333ee9D6bF30629A098B9ed](#))

2. EIGEN Strategy

(스테이킹 계약: [0xaCB55C530Acdb2849e6d4f36992Cd8c9D50ED8F7](#))

3. ether.fi Strategy

(스테이킹 계약: [0x7079A4277eAF578cbe9682ac7BC3Efff8635ebBf](#))

각 전략에는 고유한 가중치(weight)가 부여된다. 운영자의 전체 투표권은 각 스테이킹 계약 내 스테이킹 금액에 해당 가중치를 곱한 값들의 합으로 계산된다. 즉, 아래의 수식으로 표현할 수 있다.

$$\text{Voting Power} = \sum_{i=1}^n (\text{Staked Amount}_i \times \text{Weight}_i)$$

이 가중치는 RED 스테이커에게 유리하게 조정되어, 그들이 투표권과 보상의 대부분을 차지하도록 설계되어 있다.

네트워크의 균형을 유지하고 중앙화를 방지하기 위해, 레드스톤 AVS는 **운영자별 투표권 상한(Voting Power Cap)**을 적용한다. 즉, 실제 투표권은 다음 중 더 작은 값으로 설정된다:

- 위의 가중치 기반 스테이킹 공식으로 계산된 값
- 미리 정의된 최대 투표권 한도

이 메커니즘은 단일 운영자가 과도한 스테이킹을 통해 검증 결과를 지배하는 것을 방지한다.

tip

특정 운영자의 투표권은 [AttestationCenter](#) 컨트랙트에서 다음 함수를 통해 확인할 수 있다.

```
function votingPower(address _operator)
```

보상 계산(Reward calculation)

레드스톤 AVS에서는 운영자가 수행한 검증 및 투표(task vote)마다 보상이 지급된다. 태스크가 완료되면, Aggregator는 태스크 결과와 모든 운영자 투표를 AttestationCenter 스마트 컨트랙트로 전송한다. 이 시점에서 검증에 참여해 투표를 제출한 모든 운영자는 보상을 받는다.

important

레드스톤 AVS 참여 보상은 RED 토큰으로 지급된다.

각 운영자의 실제 보상 금액은 FeeCalculator라는 별도의 스마트 컨트랙트에서 계산된다. 이 컨트랙트는 각 태스크에 할당된 보상 금액을 보관하고, 투표 시점의 투표권 비율에 따라 참여한 운영자에게 분배한다.

운영자별 보상은 **투표권 비율에 비례**하여 계산되며, 아래 수식을 따른다.

$$\text{Reward Share}_i = (\text{Voting Power}_i / \text{Total Voting Power}) \times \text{Task Reward}$$

단, 레드스톤이 직접 운영하는 노드는 보상 계산에서 제외된다.

tip

각 운영자에게 누적된 보상은 AttestationCenter 컨트랙트에 저장된다. 운영자는 아래 함수를 사용해 자신에게 지급될 대기 보상을 조회할 수 있다.

```
function getOperatorPaymentDetail(uint256 _operatorId)
```

현재 FeeCalculator 주소는 SetFeeCalculator 이벤트를 통해 확인할 수 있다.

(topico:

0x83b9ee7f260088fdd4ee12aa07fa7daebc115d796b6bfb55bfb0fc839bccff2d)

보상 분배(Rewards distribution)

AVS 컨트랙트에 누적된 보상은 **매주(weekly)** 운영자와 스테이커 모두에게 분배된다. 이 분배는 ELIP-001에 명시된 EigenLayer Rewards v2 메커니즘을 통해 수행된다.

이 과정은 EigenLayer의 Rewards Coordinator 컨트랙트와의 통합을 포함하며, 다음 데이터를 제출한다:

- 각 운영자별로 수집된 보상 금액
- 지원되는 스테이킹 전략 목록 및 해당 가중치
- 보상이 발생한 기간(time period)

이 데이터를 기반으로 다음 단계가 수행된다:

1. 운영자 수수료 차감(Operator fee deduction)

각 운영자는 자신이 유지할 보상의 비율(0%~100%)을 정의할 수 있다.

기본값은 10%이며, 차감된 부분은 운영자에게 서비스 수수료로 직접 지급된다.

2. 스테이커 보상 분배(Staker reward distribution)

나머지 보상 금액은 해당 운영자의 스테이커들에게 분배된다.

운영자가 자신의 자산을 스테이킹한 경우, 자신도 스테이커로서 보상에 포함된다.

분배는 각 스테이커의 스테이킹 금액과 스테이킹 전략의 가중치에 비례해 계산된다.

보상 계산은 **오프체인(off-chain)** 에서 수행되며, 이후 **온체인(on-chain)** 에 제출되어 검증된 후 청구 가능 상태가 된다. 이 과정의 세부 사항은 EigenLayer [개발자 가이드](#)를 참고하라.

분배가 완료되면, 운영자와 스테이커는 [EigenLayer UI](#)를 통해 보상을 청구할 수 있다.

슬래싱(Slashing)

현재 레드스톤 AVS는 슬래싱 메커니즘을 구현하지 않는다. 향후 도입될 경우, 초기 단계에서는 **운영자의 가용성(uptime)** 이나 **접속 유지**보다는 **데이터 검증의 정확성(correctness)** 에 초점을 맞출 예정이다.

슬래싱 정책이나 집행과 관련된 모든 변경 사항은 참여자들이 사전에 준비할 수 있도록 **충분한 사전 공지**를 통해 투명하게 안내될 것이다.

AVS 운영자 실행(Running AVS Operator)

레드스톤 AVS 생태계의 운영자가 되는 것은 최첨단 탈중앙화 오라클 네트워크의 핵심 역할을 수행할 수 있는 특별한 기회이다. 운영자는 블록체인 전반의 스마트 컨트랙트 및 탈중앙화 애플리케이션(dApp)에 제공되는 데이터의 정확성과 보안성 유지에 직접 기여하게 된다. 이 혁신적인 시스템의 무결성을 보장하는 동시에, 참여를 통해 보상을 획득할 수도 있다.

info

아래의 지침은 이더리움 [Holesky](#) (L1) 테스트넷 환경을 기준으로 하며, 이 환경에는 EigenLayer 및 Othentic 보상 관련 컨트랙트가 배포되어 있다. 또한 L2 Base [Sepolia](#) 네트워크에는 어테스터 서명을 검증하고 현재 가격과 타임스탬프를 저장하는 컨트랙트가 배포되어 있다.

레드스톤 AVS 네트워크에서 운영자가 되기 위해서는 다음 네 가지 단계를 순서대로 따라야 한다.

1단계: 운영자 화이트리스트 등록(Getting Whitelisted as an Operator)

운영자로 등록하기 위해서는 먼저 **화이트리스트(operator whitelist)** 에 추가되어야 한다. 현재는 **1단계(Phase 1)** 로, 선택된 일부 운영자만이 화이트리스트에 등록된다. 이 절차는 향후 **2단계(Phase 2)** 에서 더 많은 참여가 가능하도록 변경될 예정이다.

2단계: 운영자 등록(Registering as an Operator)

화이트리스트에 등록된 후, 운영자가 되기 위한 다음 단계는 **AVS와 EigenLayer 시스템에 등록**하는 것이다. 이 등록 절차는 운영자의 신원을 설정하고 레드스톤 AVS 네트워크에서 참여할 수 있도록 하는 데 필수적이다.

등록하려면 터미널에서 다음 명령을 실행한다:

```
docker run --platform linux/amd64 -it public.ecr.aws/y7v2w8b2/avs-othentic-
client:e54c10e1 \
  operator register \
  --l1-chain holesky
```

명령 실행 후 다음 정보를 입력해야 한다:

- **운영자 개인 키(Operator's Private Key) (Controller Key)**: Othentic 공유 보안 프로토콜에 가입하는 데 사용되는 키이며, 리스테이커(restaker)가 스테이킹 자산을 위임(delegate)하는 계정이다.
- **합의용 별도 키 사용 여부(Use a different private key for Consensus?) (Consensus Key)**: 합의의 메시지 서명에 별도의 키를 사용하려면 “Yes”를 선택하고, 동일한 키를 사용할 경우 “No”를 선택한다.
- **AVS 거버넌스 컨트랙트 주소(AVS Governance Contract Address)** 아래 주소를 입력한다:
0x58ab5f17D13E56F6048890BBC7Ba3E44d00A3ED8
- **보상 수령 주소(Rewards Receiver Address)** (선택사항) 입력하지 않으면 기본값으로 운영자 주소가 사용된다.
- **EigenLayer 미등록 시 추가 정보 제공** 운영자 이름, 설명, 웹사이트, 로고 URL, 트위터 프로필 등의 세부 정보를 입력해야 한다.

important

등록 후, 레드스톤 팀에 알려주어야 한다. 그러면 해당 운영자를 **태스크 정의(Task Definition)**에 명시된 제한된 어테스터 목록(restricted attesters list)에 추가할 수 있다.

3단계: 지원되는 전략에 스테이킹 예치(Depositing Stake on a Supported Strategy)

EigenLayer에서 운영자로 등록한 후에는 하나 이상의 지원되는 스테이킹 전략(Strategy)에 스테이크를 예치해야 한다. 이 스테이크는 운영자가 네트워크에 재정적 기여와 책임을 지닌다는 것을 보장한다.

info

0 이상의 스테이크를 보유해야 레드스톤 AVS 생태계에서 투표권(Voting Power)을 얻을 수 있다. 스테이크가 0이면 운영자는 비활성 상태가 되어 합의(Consensus)와 거버넌스에 참여할 수 없다.

info

화이트리스트 등록 후, 일부 **RedStone Beta Points (RBP)**가 지급되며, 이는 리스테이킹(Restaking)에 사용할 수 있다.

아래는 현재 지원되는 스테이킹 전략 목록이다. 각 전략에 대한 **Deposit 명령어**를 사용하거나, [EigenLayer 애플리케이션](#)을 통해 직접 리스테이킹을 수행할 수 있다.

EIGEN 전략(EIGEN Strategy)

전략 주소(Strategy Address): [0x43252609bffa13dFe5e057097f2f45A24387a84](#)

리스테이킹 페이지(Restake): [EIGEN](#)

예치 명령어(Deposit Command):

```
docker run --platform linux/amd64 -it public.ecr.aws/y7v2w8b2/avs-othentic-
client:e54c10e1 \
  operator deposit \
  --l1-chain holesky \
  --staking-contract EIGENETH
```

RBP (RedStone Beta Points) 전략

테스트넷에서만 임시로 사용되는 스테이킹 전략이다.

전략 주소(Strategy Address): [0xb489145e027eE17A33c145aE498c920F89DF640A](#)

리스테이킹 페이지(Restake): [RBP Token](#)

예치 명령어(Deposit Command):

```
docker run --platform linux/amd64 -it public.ecr.aws/y7v2w8b2/avs-othentic-
client:e54c10e1 \
  operator deposit \
  --l1-chain holesky \
  --staking-contract-address 0xb489145e027eE17A33c145aE498c920F89DF640A
```

4단계: 설정 파일 준비(Preparing the Configuration File)

다음 단계는 운영자가 레드스톤 AVS 네트워크 내에서 올바르게 작동하기 위해 필요한 필수 매개변수를 정의하는 구성 파일을 작성하는 것이다. 이 구성 파일은 `.env`라는 이름으로 저장되며, `.env.example` 파일의 구조를 따른다.

아래 표는 각 필드의 설명과 입력해야 하는 항목을 정리한 것이다:

매개변수(Parameter)	설명(Description)
PRIVATE_KEY	Consensus Key — 합의 서명을 위한 개인 키
OPERATOR_ADDRESS	Controller Key 의 공개 주소 (두 키가 동일할 경우 선택사항)
L1_RPC / L2_RPC	각각 L1 (Ethereum Holesky 테스트넷) 및 L2 (Base Sepolia 테스트넷)의 RPC 엔드포인트 주소
AVS_GOVERNANCE_ADDRESS	AVS 거버넌스 컨트랙트 주소 (0x58ab5f17D13E56F6048890BBC7Ba3E44d00A3ED8)
ATTESTATION_CENTER_ADDRESS	Attestation Center 컨트랙트 주소 (0x6af9B9272fc72CaC55ccDF6c2BC2c5703a65a187)
ANNOUNCED_ADDRESSES	공개 가능한 노드 주소 — 아래 프레임에서 상세 설명 참고

tip

- # 주석이 달린 항목은 운영자 설정에 따라 적절한 값을 입력해야 한다.
- .env 파일 대신 시스템 전역 환경 변수로 설정할 수도 있다.

tip

만약 Docker 컨테이너의 IP가 외부에서 접근 가능한 공용 IP라면, `docker-compose.yml` 파일의 `--announced-addresses` 및 `$ANNOUNCED_ADDRESSES` 매개변수를 주석 처리한 상태로 유지한다. 그러나 외부에서 접근 가능한 노드가 아닌 경우, RedStone의 Aggregator 노드가 연결할 수 있도록 멀티어дрес스(Multi-address) 형태로 공개 주소를 제공해야 한다.

자세한 형식은 [libp2p 주소 규격 문서](#)를 참고한다.

- 노드가 IP를 통해 접근 가능하다면 `/ip4/` 로 시작하는 형식을 사용.
- 도메인 이름으로 접근 가능한 경우 `/dns/` 로 시작하는 형식을 사용.
정적 IP를 보유한 경우 `/ip4/` 항목도 함께 포함해야 다양한 탐색 메커니즘과의 호환성을 유지할 수 있다.

`<peer_id>` 값은 첫 실행 시 로그에서 확인 가능하다. `docker-compose.yml` 파일에서 `--announced-addresses`를 주석 처리한 상태로 노드를 실행한 후, 로그에서 "Listening on the following addresses" 구문을 검색한다. `peer_id`는 "12D3K" 로 시작하는 문자열이다.

`peer_id`를 확인한 후, .env 파일에서 관련 변수를 활성화하고 `<peer_id>`, `<ip_address>` 또는 `<domain_name>`을 입력한다. 이후 `docker-compose.yml` 파일 내 `--announced-addresses` 및 `${ANNOUNCED_ADDRESSES}` 항목의 주석을 해제한다.

만약 인터넷에서 직접 접근할 수 없는 Docker 컨테이너라면, 여전히 운영자로 참여할 수 있으나 네트워크 연결이 간헐적으로 끊길 수 있으며, 주기적으로 노드를 재시작해야 할 수 있다.

추가적인 내용은 Othentic [공식 문서](#)를 참고한다.

5단계: 운영자 실행(Running the Operator)

레드스톤 AVS 네트워크 내에서 운영하려면 두 개의 Docker 이미지가 필요하다: [Attester](#) 및 [Validation API](#). 이 두 이미지는 [Docker Compose 구성 파일](#)을 통해 실행할 수 있다. 운영자를 시작하려면 `docker-compose.yml` 파일이 있는 디렉터리에서 다음 명령어를 실행한다:

```
docker compose up
```

이 명령어를 실행하면 Docker Compose 파일에 정의된 대로 **Attester**와 **Validation API** 서비스가 동시에 실행된다.

tip

[docker-compose.yml](#) 파일에는 영구 데이터 저장을 위한 볼륨 마운트가 포함되어 있다.

volumes:

- .othentic:/app/.othentic

이 디렉터리는 [Othentic 문서](#)에 따라 영구 데이터를 저장하는 데 사용된다. 호스트 시스템에서 이 볼륨의 경로를 환경에 맞게 수정할 수 있다.

tip

위의 두 Docker 이미지는 반드시 Docker Compose를 통해서만 실행할 필요는 없다. 운영 환경에 따라 Kubernetes, Docker Swarm 등 다른 방식으로 배포할 수 있다.

운영자 업데이트(Updating the Operator)

운영자 코드를 업데이트하려면 다음 단계를 수행한다:

1. [docker-compose.yml](#) 파일이 있는 디렉터리(복제된 [Operator](#) 저장소)로 이동
2. `git pull` 실행
3. `docker compose restart` 실행

메트릭 및 모니터링(Metrics and monitoring)

모니터링과 메트릭 수집을 활성화하려면 운영자 노드를 다음 매개변수와 함께 실행한다:

```
--metrics --metrics.port <port>
```

자세한 내용은 Othentic [문서](#)를 참고한다.

검증 API(Validation API)

레드스톤 AVS 네트워크는 Othentic 표준 검증 API를 사용하며, 이 API는 `POST /task/validate` 엔드포인트를 통해 호출된다. 전체 사양은 Othentic [문서](#)에서 확인할 수 있다.

운영자 등록 해제(Unregistering the Operator)

레드스톤 AVS 네트워크에서 더 이상 운영하지 않으려면 다음 명령어로 운영자 등록을 해제할 수 있다:

```
docker run --platform linux/amd64 -it public.ecr.aws/y7v2w8b2/avs-othentic-
client:e54c10e1 \
operator unregister \
--l1-chain holesky
```

이 과정에서 개인 키(Private Key)와 AVS 거버넌스 컨트랙트 주소 (0x58ab5f17D13E56F6048890BBC7Ba3E44d00A3ED8)를 제공해야 한다.

하드웨어 요구사항(Hardware Requirements)

- **CPU:** 1 vCPU
- **RAM:** 2 GB
- **Storage:** 1 GB
- **인터넷:** 안정적이고 빠른 네트워크 연결 권장

자주 묻는 질문(FAQ)

Q. Docker 컨테이너 실행 시 다음 오류가 발생합니다: Failed to open datastore at .othentic/data/p2p: Error: Database is not open. 어떻게 해결해야 하나요?

이 오류는 일반적으로 예상된 데이터 디렉터리가 존재하지 않거나 잘못된 소유권(ownership)을 가진 경우 발생한다. `docker compose`를 실행하는 디렉터리 안에 `.othentic` 폴더가 존재하는지 확인하고, 해당 폴더의 소유자가 `1000:1000`인지 확인한다.

- 존재하지 않는다면 아래 명령어로 폴더를 생성하고 소유권을 변경한다.
`mkdir .othentic`
`sudo chown 1000:1000 .othentic`
- 만약 `docker-compose.yml`의 `volumes` 항목에서 다른 폴더를 지정한 경우, 위 명령어에서 해당 경로를 맞게 수정해야 한다.

토큰(Token)

경제적 보안을 위한 스테이킹(Staking For Economic Security)

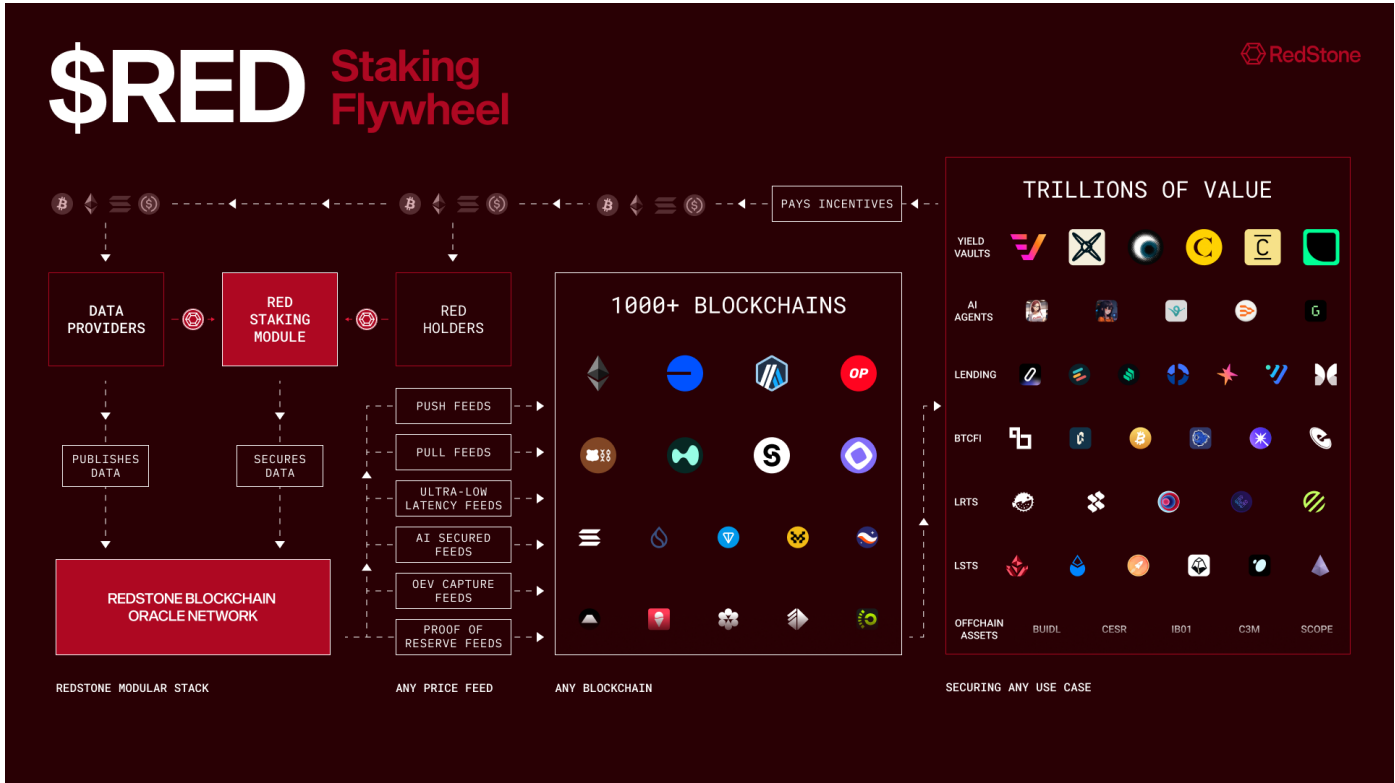
오라클이 장기적으로 생존하고 DeFi를 보호하기 위해서는 지속 가능한 토크노믹스(tokenomics)가 필수적이다. RED는 혁신적인 가치 축적(value accrual) 메커니즘을 갖춘 유틸리티 토큰으로 설계되었으며, 진정한 의미의 지속 가능한 오라클 경제 구조를 제시한다.

레드스톤의 EigenLayer 기반 **Actively Validated Service (AVS)** 를 활용하여, RED 스테이킹은 레드스톤 오라클 스택에 강력한 경제적 보안 계층을 추가한다. 스테이킹된 RED뿐 아니라 EigenLayer에 예치된 수십억 달러 규모의 자산을 활용해 추가적인 보안을 제공할 수 있다.

데이터 공급자(data provider)와 토큰 보유자(token holder)는 모두 RED를 스테이킹할 수 있다.

- **데이터 공급자**는 레드스톤의 모듈형 오라클 네트워크에 데이터를 제공한다.
- **토큰 보유자**는 레드스톤 AVS에 직접 스테이킹함으로써 네트워크의 보안을 강화한다.

RED 스테이커는 ETH, BTC, SOL, USDC 등 주요 자산으로 지급되는 보상을 수백 개의 블록체인에서 레드스톤 데이터를 사용하는 사용자로부터 받게 된다.



info

본 문서는 정보 제공 목적으로만 작성되었으며, 토큰 매도 제안, 매수 권유, 또는 투자 유도 목적의 문서가 아닙니다. 법률, 투자, 재정, 세무 또는 기타 전문적 조언으로 해석되어서는 안 되며, 반드시 자격 있는 전문가의 조언이나 직접적인 조사를 통해 판단해야 합니다.

본 문서에 포함된 수치 및 전망은 가정 또는 추정치로, 실제 결과와 다를 수 있으며 보장되지 않는다.

RedStone Distributed Data Association 및 그 자회사는 본 정보의 완전성, 신뢰성, 정확성에 대해 어떠한 명시적 또는 묵시적 보증도 하지 않는다.

본 문서의 모든 내용은 사전 통보 없이 변경될 수 있다.

공급(Supply)

네트워크(Network)	주소(Address)
Ethereum	0xc43c6bfeda065fe2c4c11765bf838789bdobb5de
Base	0x4eB92702bA4cfbF80561BAd64D89C706Ac824960

레드스톤의 블록체인 오라클 기술은 2023년 1월부터 메인넷에서 안정적으로 운영되어 왔으며, 다양하고 신뢰성 높은 서비스를 통해 사용자 기반을 확보했다. 이제 레드스톤 분산 데이터 협회(RedStone Distributed Data Association)는 RED 토큰을 도입하여 시스템의 탈중앙화를 더욱 강화하고 레드스톤 생태계의 장기적 비전을 실현하고자 한다.

레드스톤의 핵심 미션은 “모든 체인, 모든 데이터, 모든 용도에 맞는 빌더 지원”이다. RED 토큰 출시(Token Generation Event, TGE)를 앞두고 RED의 유틸리티와 토큰 이코노믹스를 아래와 같이 소개한다.

항목(Category)	세부사항(Details)
토큰 심볼(Ticker)	RED
최대 공급량(Max Supply)	1,000,000,000 RED
커뮤니티 성장 비율(Community Growth Allocation)	48.3%
초기 유통량(Float at TGE)	28%
토큰 표준(Token Standard)	Ethereum Network: ERC-20
락업 비율(Lock Up)	초기 RED의 72%가 잠금 상태이며, 4년에 걸쳐 토큰 릴리즈 일정에 따라 점진적으로 해제된다.

info

본 문서는 정보 제공 목적이며, 토큰 매수 권유나 판매 제안이 아니다. 법적, 재정적, 세무적, 투자적 조언으로 간주되지 않으며, 모든 수치는 추정치로 실제 결과와 다를 수 있다.

RedStone Distributed Data Association 및 그 자회사는 본 문서의 내용에 대해 어떤 명시적/묵시적 보증도 하지 않는다. 본 문서의 정보는 사전 고지 없이 변경될 수 있다.

토큰 분배(Token Distribution)

RED의 최대 공급량은 **1,000,000,000 (10억)** 개이며, 초기 유통량은 **30%**이다. RED는 이더리움 메인넷에서 **ERC-20 토큰**으로 구현되며, 토큰 생성 이벤트(Token Generation Event, TGE) 이후에는 **Wormhole Native Token Transfer** 표준을 통해 솔라나(Solana), 베이스(Base), 기타 네트워크에서도 이용 가능해질 예정이다.

전체 RED 토큰의 약 절반(48.3%)은 레드스톤 생태계 및 커뮤니티를 위해 할당되며, 세부 분류는 **커뮤니티 & 제네시스(Community & Genesis)**, **에코시스템 & 데이터 제공자(Ecosystem & Data Providers)**, 그리고 **프로토콜 개발(Protocol Development)** 범주로 나뉜다.

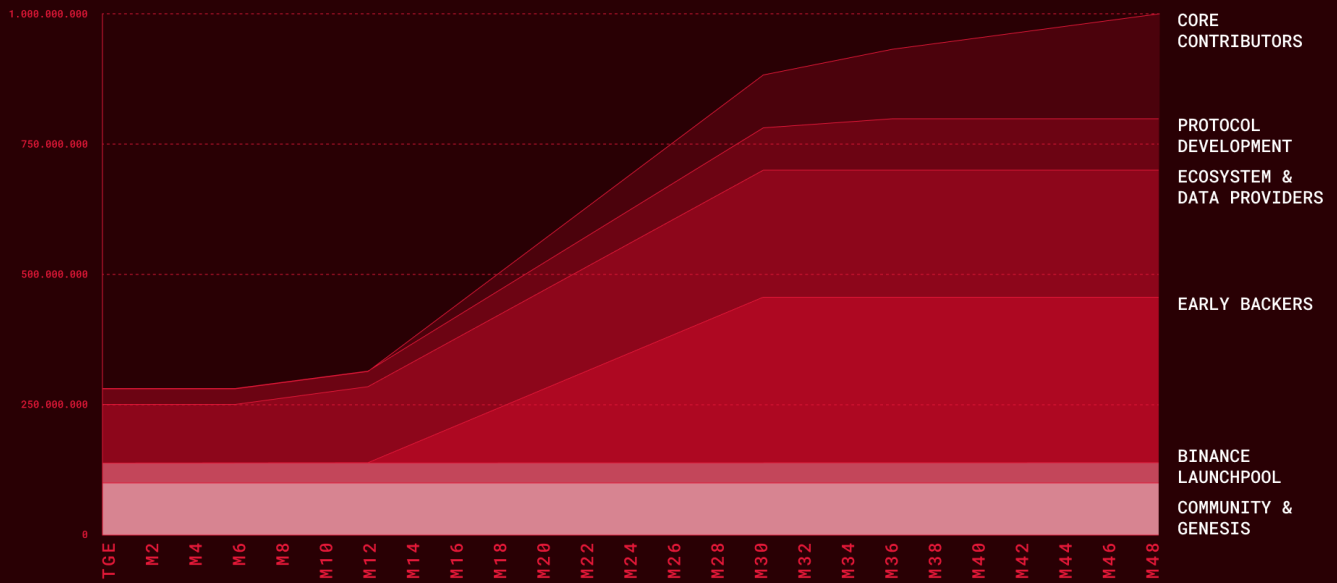
RED 분배 카테고리(RED Distribution Categories)

카테고리(Category)	총 비율(Total)
커뮤니티 & 제네시스(Community & Genesis)	10.00%
프로토콜 개발(Protocol development, 부분 락업)	10.00%
핵심 기여자(Core Contributors, 락업)	20.00%
바이낸스 런치풀(Binance Launchpool)	4.00%
에코시스템 & 데이터 제공자(Ecosystem & Data Providers, 부분 락업)	24.30%
초기 투자자(Early Backers, 락업)	31.70%
총계(Total)	100%

RED 언락 일정(RED Unlock Schedule)

\$RED Token Unlock Schedule

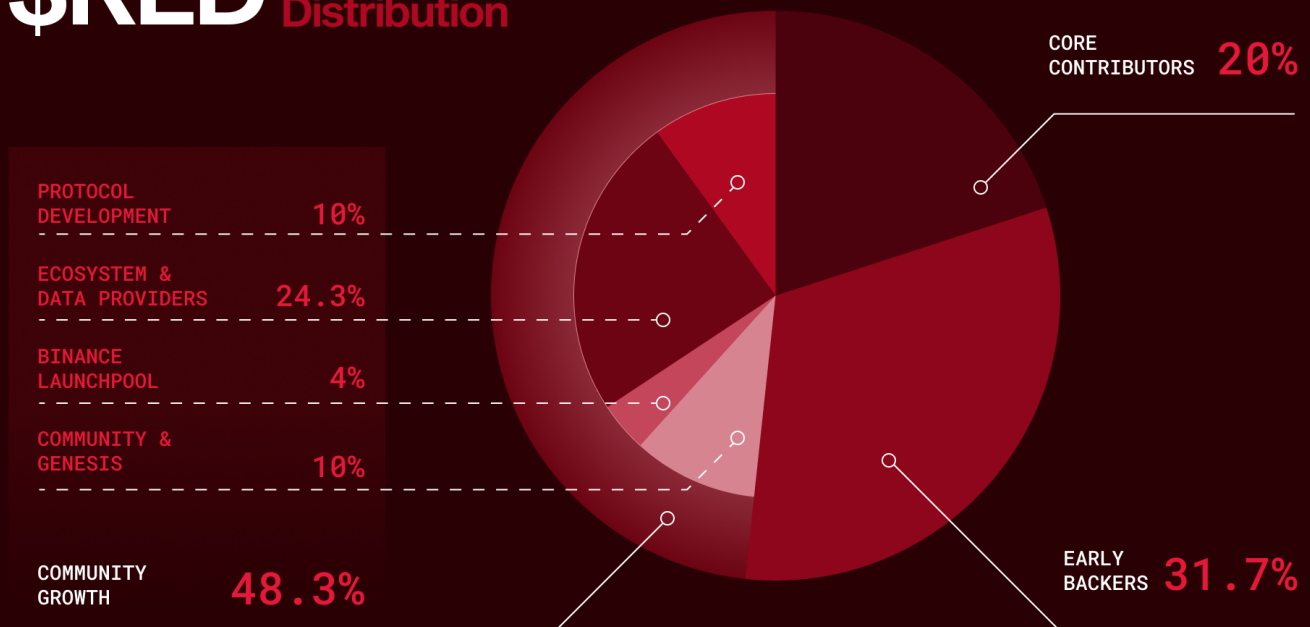
RedStone



RED 분배 차트(RED Distribution Chart)

\$RED Token Distribution

RedStone



커뮤니티 & 제네시스 (Community & Genesis, 10%)

전체 RED 토큰의 10%는 레드스톤 커뮤니티에 배분되며, 초기 청구(Claim) 활동을 통해 배포된다. 이 커뮤니티 & 제네시스 할당은 레드스톤 토큰노믹스의 핵심으로, 프로토콜의 장기적 성공에 있어 커뮤니티가 수행하는 중요한 역할을 인정한다. 이 토큰들은 에어드롭(Airdrop) 형태로 신중히 설계되어 배포되며, 초기 지지자, 파트너, 레드스톤의 성장에 기여한 적극적 참여자들에게 보상으로 지급된다. 이 전략은 생태계의 초기 참여자와 지속적 기여자들이 공정하게 대표될 수 있도록 설계되었다.

이 중 4.5%는 DRILL 캠페인(Develop, Reinforce, Innovate, Launch & Learn) 을 통해 배포되며, 해당 전용 멀티시그 주소는 0xBA544abd1b34C4a337E3F3Cbe2A390e061031FF7 이다.

프로토콜 개발 (Protocol Development, 10%)

전체 RED 토큰의 10%는 프로토콜 개발에 할당되어, 레드스톤의 장기적 지속 가능성과 성장을 보장한다. 이 자금은 핵심 연구 및 개발, 개발자 리소스, 운영 효율성 강화를 지원한다.

레드스톤 분산 데이터 협회(RedStone Distributed Data Association)는 이 자원을 전략적으로 배분하여 프로토콜의 성장을 촉진하고 점진적인 탈중앙화를 추진한다.

핵심 기여자 (Core Contributors, 20%)

전체 RED 토큰의 20%는 레드스톤 생태계의 핵심 기여자들에게 배정된다. 이는 프로토콜의 기반을 구축하고 장기적 성장을 주도할 인물들을 인정하기 위한 것이다. 이 할당은 보안, 엔지니어링, 제품 개발, 커뮤니티, 마케팅, 전략, 사업 개발, 운영 등 여러 분야의 주요 기여자들의 지속적인 헌신을 보장한다.

해당 카테고리의 모든 토큰은 락업(Locked) 상태이며, 기여자의 이해관계를 프로토콜의 장기적 성장 및 성공과 일치시키기 위한 전략적 베스팅 일정(vesting schedule) 을 따른다.

바이낸스 런치풀 (Binance Launchpool, 4%)

전체 RED 토큰의 4%는 바이낸스 런치풀(Binance Launchpool) 에 할당되어, 바이낸스와 레드스톤 커뮤니티 간 협력 관계를 강화하기 위한 전략적 파트너십 기반을 형성한다.

에코시스템 & 데이터 제공자 (Ecosystem & Data Providers, 24.3%)

전체 RED 토큰의 24.3%는 에코시스템 & 데이터 제공자(Ecosystem & Data Providers) 부문에 할당된다.

- **에코시스템(Ecosystem)** 부분은 레드스톤의 혁신과 커뮤니티 성장을 촉진하는 데 사용되며, 빌더, 혁신가, 커뮤니티 리더를 대상으로 한 보조금(grants)과 인센티브 프로그램을 통해 지원한다. 또한 커뮤니티 중심 프로젝트, 교육 프로그램, 시장 확대 및 채택 촉진 활동을 지원하며, 레드스톤 생태계 확장에 실질적 가치를 더하는 프로젝트에 중점을 둔다. 새로운 제품 및 서비스가 임계점(critical mass) 에 도달할 때까지 도입을 촉진하는 역할도 수행한다.
- **데이터 제공자(Data Providers)** 부문은 레드스톤 오라클 인프라에 데이터를 제공하는 참여자에게 보상한다. 이는 고품질, 적시성 높은 데이터 제공을 보장하기 위한 인센티브 구조로, 새로운 시장 페어(pair) 및 변

화하는 데이터 수요에 대응하도록 전략적 보조금과 인센티브를 제공한다. 이를 통해 최고 수준의 데이터 품질을 유지하며, 안정적인 데이터 공급 생태계를 형성한다.

해당 카테고리의 모든 토큰은 **구조화된 릴리즈 일정(release schedule)**을 따르며, 레드스톤의 장기적 비전과 일치하는 지속 가능한 생태계 성장 및 데이터 제공자 다양성을 지원한다.

초기 투자자 (Early Backers, 31.7%)

전체 RED 토큰의 31.7%는 **초기 투자자(Early Backers)**에게 배정된다. 이들은 레드스톤 개발 단계에서 핵심적인 재정적, 자문적, 전략적 지원을 제공했으며, 그들의 산업 전문성과 인프라 역량은 프로토콜의 기반을 강화하는 데 기여했다.

이 카테고리의 토큰은 **락업(Locked)** 상태이며, 초기 투자자의 이해관계를 레드스톤의 장기적 성공과 일치시키기 위한 **전략적 베스팅 일정**을 따른다. 이를 통해 프로토콜의 발전 과정에서 지속적인 지원과 참여를 유도한다.

info

본 문서는 정보 제공 목적이며, 토큰 매도 제안, 매수 권유, 투자 유도 목적의 문서가 아니다. 법률, 투자, 재정, 세무 또는 기타 전문적 조언으로 간주되어서는 안 되며, 반드시 자격 있는 전문가의 조언이나 자체 연구를 통해 판단해야 한다. 본 문서에 포함된 수치 및 전망은 추정치로, 실제 결과와 다를 수 있으며 보장되지 않는다. RedStone Distributed Data Association 및 그 자회사는 본 문서의 완전성, 신뢰성, 정확성에 대해 어떠한 명시적 또는 묵시적 보증도 하지 않는다. 본 문서의 모든 내용은 사전 통보 없이 변경될 수 있다.