CCIP 1.6 Documentation

Glossary

- **Blessing:** An explicit approval from an RMN node on a given commitment
- **Curse:** A warning - usually about a chain, but could be any arbitrary subject - that something is broken and shouldn’t be used, e.g. a specific chain stopped producing new blocks
- **DON:** A *decentralized oracle network*, the set of Chainlink nodes coordinating to perform a specific action
- **Home chain:** The chain that `RMNHome` and `CCIPHome` contracts are deployed to
- **Job / Job Spec:** a configuration file that each Chainlink node uses to configure its participation in the DON running CCIP
- **Lane:** A message channel between two chains, ex Arb ⇔ OP and Arb ⇔ Polygon
- **NOP / Node Operator:** the person or organization behind a single node in a DON
- **OCR:** consensus protocol designed specifically for Chainlink nodes
- **Report:** the payload produced by OCR, signed by DON participants, and relayed to chain; there are two types of reports in CCIP: commit reports and exec (execute) reports
- **Keystone**: A new form of price reporting
- **Capability:** A specific function or service that a node can perform or support within the network, managed and tracked by the `CapabilitiesRegistry`
- **RMN:** the *risk management network*, a separate conglomerate of nodes specifically dedicated to observing and validating commit roots in CCIP

Background

For the most comprehensive background on Chainlink CCIP, see our public docs:

Chainlink Documentation | Chainlink Documentation

In prior CCIP versions (1.5 and below), a lane is established by a uni-directional connection between two sets of smart contracts residing on different chains. Each connecting set of contracts is considered to be a lane and each unidirectional lane operates with: 1 contract on the source chain (`EVM2EVMOnRamp`) and 2 contracts on the destination chain (`CommitStore` + `EVM2EVMOffRamp`).

For example, 2 users looking to transfer tokens from the same chain to 2 different chains will use different lanes and therefore 2 different sets of contracts:

- User 1 sending tokens from Ethereum → Polygon will use:
  - `EVM2EVMOnRamp` on Ethereum
  - `EVM2EVMOffRamp` on Polygon
  - `CommitStore` on Polygon
- User 2 sending tokens from Ethereum → Arbitrum will use:
  - `EVM2EVMOnRamp` on Ethereum

  - `EVM2EVMOffRamp` on Arbitrum
  - `CommitStore` on Arbitrum

Also, in 1.5 and below, each Risk Management Network node calls the Risk Management contract to *"bless"* the committed Merkle root. When there are enough blessing votes, the root becomes available for execution. In case of anomalies, each Risk Management node calls the Risk Management contract to *"curse"* the system. If the cursed quorum is reached, the Risk Management contract is paused to prevent any CCIP transaction from being executed.

Finally, prior CCIP versions have EVM specific logic hence the `EVM2EVMOn/OffRamp` names, making them incompatible with non EVM chains.

Goals

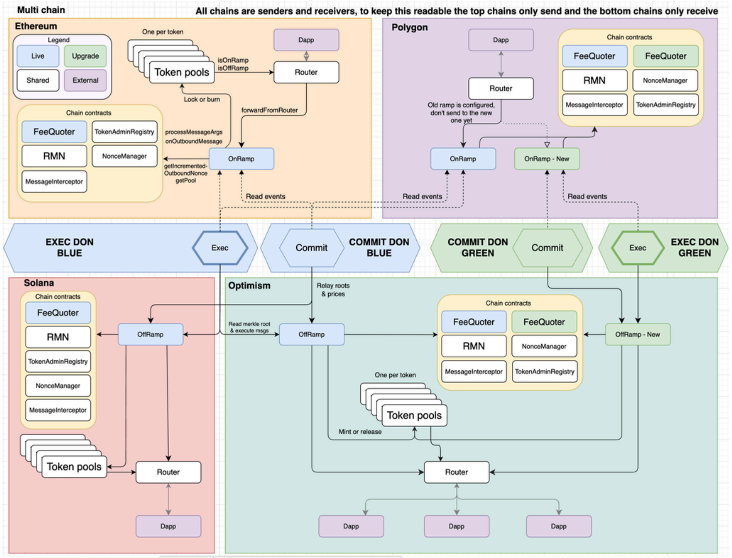
The main goal of CCIP v1.6 is to address the current limitations with regards to scalability by moving from a single on/off ramp per lane, to a single on/off ramp per chain, shared across lanes and chain families (sometimes referred to as a multi-lane architecture). By making the contracts non lane specific in v1.6, there are no more contracts to be deployed for new unidirectional lane, only configs must be set.

Another goal is to convert CCIP v1.6 contracts to be chain family agnostic (i.e. not tied to EVM), enabling bridging across different chain families. The major identified differences between chains are the below:

- Addresses may be different than 20 bytes.
- The resources used in transaction execution may be billed differently across families (for example Aptos and ZKsync charge for state changes separately).

There is still a desire to perform destination chain-specific validations when users send messages, there must be some family specific code on-chain. Therefore, a new chain family may require breaking changes, but the goal is to minimize it.

For high level contract interactions please refer to below diagram:



CCIP v1.6 Contract Interactions

Scope:

The scope of this audit is confined to the core CCIP contracts outlined below, *token pool contracts are out of scope.*

OnRamp

The CCIP v1.6 `OnRamp` has been changed in the following ways:

1. Multiple lanes support: all lane specific variables can now be configured directly either on the `OnRamp` or the `FeeQuoter` instead of having to deploy a new contract
2. Chain family agnostic logic: prior ramp contracts were EVM specific, v1.6 contracts have been designed to be compatible with many chain families.
3. Outsourced fee and chain family specific logic to the `FeeQuoter`: this change makes the `OnRamp` destination chain family agnostic - future chain families would typically require only upgrading the `FeeQuoter` contract.
4. Outsourced nonce management logic: sender's nonce management are now managed by the `NonceManager` contract. This makes nonce preservation between upgrades significantly more efficient.
5. Outsourced rate limiting logic: rate limiting is now optional and managed by a hook `IMessageInterceptor` contract. This interceptor could do things besides rate limiting as well.
6. Outsourced billing logic: node operators payment logic has been outsourced to a separate `FeeAggregator` contract (not in scope of this audit).

Multi lane support

In order to support multiple lanes, some state variables have been updated. Destination chain specific variables are now stored in a `DestChainConfig` struct. Destination chain specific pricing configs have been moved to the `FeeQuoter` (previously named `PriceRegistry`).

```
/// @dev Struct to hold the configs for a destination chain
/// @dev sequenceNumber, allowListEnabled, router will all be packed in 1 slot
struct DestChainConfig {
  // The last used sequence number. This is zero in the case where no messages have yet been sent.
  // 0 is not a valid sequence number for any real transaction.
  uint64 sequenceNumber; // ──╮ The last used sequence number
  bool allowListEnabled; //   │ boolean indicator to specify if allowList check is enabled
  IRouter router; // ─────────╯ Local router address  that is allowed to send messages to the destination chain.
  EnumerableSet.AddressSet allowedSendersList; // The list of addresses allowed to send messages from onRamp
}
```

Nonce management

The new release also introduces a `NonceManager` to extract the sender nonce management logic from both the `OnRamp` and `OffRamp`. This brings multiple benefits including:

- Contract size reduction for the `OnRamp` and the `OffRamp`.
- Makes nonce reasoning during `OffRamp` upgrades a lot easier, given there is a single source of truth for nonces across all versions of a lane.
- `prevOnRamp` & `prevOffRamp` only need to be set once.

At the cost of:

- Extra gas cost, but reduced gas costs compared to long `prevOnRamp` or `prevOffRamp` chains previously possible.
- Extra config management (`NonceManger` permissions).
- Adding a contract to the deployment pipeline for new chains.

The `NonceManager` `getIncrementedOutboundNonce()` function is now called from the `OnRamp` `forwardFromRouter` function (if out of order execution (OOO) is disabled):

```
newMessage.header.nonce = isOutOfOrderExecution
  ? 0
  : INonceManager(i_nonceManager).getIncrementedOutboundNonce(destChainSelector, originalSender);
newMessage.extraArgs = convertedExtraArgs;
```

And the `NonceManager` will perform the nonce bump and potential callback to the previous onramp to ensure nonce sequencing:

```
function getIncrementedOutboundNonce(
  uint64 destChainSelector,
  address sender
) external onlyAuthorizedCallers returns (uint64) {
  uint64 outboundNonce = _getOutboundNonce(destChainSelector, sender) + 1;
  s_outboundNonces[destChainSelector][sender] = outboundNonce;

  return outboundNonce;
}

function _getOutboundNonce(uint64 destChainSelector, address sender) private view returns (uint64) {
  uint64 outboundNonce = s_outboundNonces[destChainSelector][sender];

  // When introducing the NonceManager with existing lanes, we still want to have sequential nonces.
  // Referencing the old onRamp preserves sequencing between updates.
  if (outboundNonce == 0) {
    address prevOnRamp = s_previousRamps[destChainSelector].prevOnRamp;
    if (prevOnRamp != address(0)) {
      return IEVM2AnyOnRamp(prevOnRamp).getSenderNonce(sender);
    }
  }

  return outboundNonce;
}
```

Rate limiting

In the existing deployment, the `OnRamp` implements an `AggregateRateLimiter` (ARL) , in v1.6 the decision was made to remove the ARL from the `onRamp` and `offRamp` to save contract size and allow us to do various other checks without having to deploy new ramps. The new approach is to configure an optional message validator and separately deploy the rate limiter validation contract.

```
address messageInterceptor = s_dynamicConfig.messageInterceptor;
if (messageInterceptor != address(0)) {
  IMessageInterceptor(messageInterceptor).onOutboundMessage(destChainSelector, message);
}
```

The only requirement of the message validator contract is that it implements the `IMessageInterceptor` interface:

```
/// @notice Interface for plug-in message hook contracts that intercept OffRamp & OnRamp messages
/// and perform validations / state changes on top of the messages. The interceptor functions are expected to
/// revert on validation failures.
interface IMessageInterceptor {
  /// @notice Common error that can be thrown on validation failures and used by consumers
  /// @param errorReason abi encoded revert reason
  error MessageValidationError(bytes errorReason);

  /// @notice Intercepts & validates the given OffRamp message. Reverts on validation failure
  /// @param message to validate
  function onInboundMessage(Client.Any2EVMMessage memory message) external;

  /// @notice Intercepts & validates the given OnRamp message. Reverts on validation failure
  /// @param destChainSelector remote destination chain selector where the message is being sent to
  /// @param message to validate
  function onOutboundMessage(uint64 destChainSelector, Client.EVM2AnyMessage memory message) external;
}
```

So it does not have to be a rate limiter contract, but as of now, the only implementation is the aggregate rate limiter. The main difference between the previous `AggregateRateLimiter` and the v1.6 `MultiAggregateLimiter` is that, similarly to the `OnRamp` , the contract supports multiple chain rate limitation (with separate buckets for the `OnRamp` and `OffRamp`).

```
/// @notice Struct to store rate limit token buckets for both lane directions
struct RateLimiterBuckets {
  RateLimiter.TokenBucket inboundLaneBucket; // Bucket for the inbound lane (remote -> local)
  RateLimiter.TokenBucket outboundLaneBucket; // Bucket for the outbound lane (local -> remote)
}

/// @notice Rate limiter token bucket states per chain, with separate buckets for incoming and outgoing lanes.
mapping(uint64 remoteChainSelector => RateLimiterBuckets buckets) s_rateLimitersByChainSelector;
```

Billing

This release extracts Node Operator (NOP) payment logic to a feeAggregator contract (not in scope).

The `OnRamp` `s_nopFeesJuels` state variable has been replaced by adding `feeValueJuels` to `CCIPMessageSent`, the CCIP message event. `feeValueJuels` is the fee amount in the smallest Link token denomination: `juels` (equal to `wei` for Ethereum). `CCIPMessageSent` will be used off-chain to calculate the total amount owed to NOPs.

```
(newMessage.feeValueJuels, isOutOfOrderExecution, convertedExtraArgs, destExecDataPerToken) = IFeeQuoter(
  s_dynamicConfig.feeQuoter
).processMessageArgs(
  destChainSelector, message.feeToken, feeTokenAmount, message.extraArgs, newMessage.tokenAmounts, tokenAmounts
);
```

OffRamp

The `OffRamp` has been changes in the follow ways:

1. Multiple lanes support: multiple source chain specific variables can now be configured directly on the `OffRamp` instead of having to deploy a new contract.
2. Chain family agnostic logic: existing ramp contracts were EVM specific contracts, v1.6 contracts have been designed to be compatible with many chain families.
3. Outsourced nonce management logic: sender's nonce management are now managed by a `NonceManager` contract
4. `CommitStore` merge: The `CommitStore` has been merged into the `OffRamp` to reduce the number of contracts to deploy and maintain. This also has some gas and safety benefits.
5. `RMN` off-chain blessing

Multi lane support

In order to support multiple lanes, some state variables have been updated. All source chain specific variables are now stored in a `SourceChainConfig` struct with the exception of execution states and committed roots which require a separate nested mapping.

Nonce management

As mentioned in the `OnRamp` nonce management section, the `OffRamp` also outsources its nonce management logic to the `NonceManager`. It does so by calling the `NonceManager` `incrementInboundNonce()` function which conditionally bumps the nonce. The nonce is only incremented if the next nonce matches the expected nonce provided by the `OffRamp` message.

If the nonce hasn't been incremented then the `OffRamp` skips the message execution:

```
if (message.header.nonce != 0) {
  if (originalState == Internal.MessageExecutionState.UNTOUCHED) {
    // If a nonce is not incremented, that means it was skipped, and we can ignore the message
    if (
      !INonceManager(i_nonceManager).incrementInboundNonce(
        sourceChainSelector, message.header.nonce, message.sender
      )
    ) continue;
  }
}
```

CommitStore merge

The `CommitStore` and `OffRamp` are now a single contract. This merge adds the following functionalities:

- Multiple OCR configurations
- Batch commits
- Batch executions

MultiOCR3

Since the `OffRamp` contract now needs to support both commit and execution reports, multiple OCR configurations are required. Hence, a new `MultiOCR3` contract was introduced.

The `OCRConfigArgs` struct (used to set new OCR configs) now has a new `ocrPluginType` field, which in the case of the `OffRamp` will help differentiate between the commit and execution plugin OCR config:

```
/// @notice OCR configuration for a single OCR plugin within a DON.
struct OCRConfig {
  ConfigInfo configInfo; //  latest OCR config
  // NOTE: len(signers) can be different from len(transmitters). There is no index relationship between the two arrays
  address[] signers; //      addresses oracles use to sign the reports
  address[] transmitters; // addresses oracles use to transmit the reports
}

/// @notice Args to update an OCR Config.
struct OCRConfigArgs {
  bytes32 configDigest; // Config digest to update to
  uint8 ocrPluginType; // ────────────────────╮ OCR plugin type to update config for
  uint8 F; //                                 │ maximum number of faulty/dishonest oracles
  bool isSignatureVerificationEnabled; // ────╯ if true, requires signers and verifies signatures on transmission verification
  address[] signers; // signing address of each oracle
  address[] transmitters; // transmission address of each oracle (i.e. the address the oracle actually sends transactions to the contract from)
}
```

And both OCR configs and oracles are now tracked in a mapping with the `ocrPulginType` as its key:

```
/// @notice mapping of OCR plugin type -> DON config
mapping(uint8 ocrPluginType => OCRConfig config) internal s_ocrConfigs;

/// @notice OCR plugin type => signer OR transmitter address mapping
mapping(uint8 ocrPluginType => mapping(address signerOrTransmiter => Oracle oracle)) internal s_oracles;
```

The `_transmit` function now also takes in an `ocrPluginType` and is called directly by either the `commit()` or the `execute()` function of the `OffRamp`.

Batch commits

As mentioned in the scalability section the lack of cross-source message batching has been one of the identified drawbacks of v1.5. In v1.6, the `OffRamp` now takes in a array of merkle roots:

```
/// @dev Struct to hold a merkle root and an interval for a source chain so that an array of these can be passed in the CommitReport.
/// @dev RMN depends on this struct, if changing, please notify the RMN maintainers.
/// @dev ineffiecient struct packing intentionally chosen to maintain order of specificity. Not a storage struct so impact is minimal.
// solhint-disable-next-line gas-struct-packing
struct MerkleRoot {
  uint64 sourceChainSelector; //     Remote source chain selector that the Merkle Root is scoped to
  bytes onRampAddress; //            Generic onramp address, to support arbitrary sources; for EVM, use abi.encode
  uint64 minSeqNr; // ───────────────╮ Minimum sequence number, inclusive
  uint64 maxSeqNr; // ───────────────╯ Maximum sequence number, inclusive
  bytes32 merkleRoot; //             Merkle root covering the interval & source chain messages
}

/// @dev Report that is committed by the observing DON at the committing phase
/// @dev RMN depends on this struct, if changing, please notify the RMN maintainers.
struct CommitReport {
  Internal.PriceUpdates priceUpdates; // Collection of gas and price updates to commit
  Internal.MerkleRoot[] merkleRoots; // Collection of merkle roots per source chain to commit
  IRMNV2.Signature[] rmnSignatures; // RMN signatures on the merkle roots
}
```

This enables transmitting multiple roots from multiple chains:

```
for (uint256 i = 0; i < commitReport.merkleRoots.length; ++i) {
  Internal.MerkleRoot memory root = commitReport.merkleRoots[i];
  uint64 sourceChainSelector = root.sourceChainSelector;

  if (i_rmn.isCursed(bytes16(uint128(sourceChainSelector)))) {
    revert CursedByRMN(sourceChainSelector);
  }

  SourceChainConfig storage sourceChainConfig = _getEnabledSourceChainConfig(sourceChainSelector);

  if (keccak256(root.onRampAddress) != keccak256(sourceChainConfig.onRamp)) {
    revert CommitOnRampMismatch(root.onRampAddress, sourceChainConfig.onRamp);
  }

  if (sourceChainConfig.minSeqNr != root.minSeqNr || root.minSeqNr > root.maxSeqNr) {
    revert InvalidInterval(root.sourceChainSelector, root.minSeqNr, root.maxSeqNr);
  }

  bytes32 merkleRoot = root.merkleRoot;
  if (merkleRoot == bytes32(0)) revert InvalidRoot();
  // If we reached this section, the report should contain a valid root
  // We disallow duplicate roots as that would reset the timestamp and
  // delay potential manual execution.
  if (s_roots[root.sourceChainSelector][merkleRoot] != 0) {
    revert RootAlreadyCommitted(root.sourceChainSelector, merkleRoot);
  }

  sourceChainConfig.minSeqNr = root.maxSeqNr + 1;
  s_roots[root.sourceChainSelector][merkleRoot] = block.timestamp;
}
```

Batch executions

Similarly to batch commits, the v1.6 `OffRamp` also enables batch executions for multiple reports from different source chains through the `execute()` function which takes in an array of `ExecutionReportSingleChain`:

```
/// @notice Report that is submitted by the execution DON at the execution phase. (including chain selector data)
/// @dev RMN depends on this struct, if changing, please notify the RMN maintainers.
struct ExecutionReportSingleChain {
  uint64 sourceChainSelector; // Source chain selector for which the report is submitted
  Any2EVMRampMessage[] messages;
  // Contains a bytes array for each message, each inner bytes array contains bytes per transferred token
  bytes[][] offchainTokenData;
  bytes32[] proofs;
  uint256 proofFlagBits;
}

/// @notice Transmit function for execution reports. The function takes no signatures,
/// and expects the exec plugin type to be configured with no signatures.
/// @param report serialized execution report
function execute(bytes32[3] calldata reportContext, bytes calldata report) external {
  _batchExecute(abi.decode(report, (Internal.ExecutionReportSingleChain[])), new GasLimitOverride[][](0));

  bytes32[] memory emptySigs = new bytes32[](0);
  _transmit(uint8(Internal.OCRPluginType.Execution), reportContext, report, emptySigs, emptySigs, bytes32(""));
}
```

RMN

RMNRemote

The fundamental RMN change in CCIP v1.6 is to move RMN blessings from on-chain to off-chain. In prior versions, RMN nodes each sent transactions to chain to explicitly bless a root *after* it was committed, and until that root was blessed, no messages contained within could be executed. In 1.6, blessings are collected *offchain* and included in the commit report. Blessings are then verified on chain as part of the `commit()` transaction.

Each `OffRamp` contains a pointer to a `RMNRemote` contract. The `RMNRemote` contract exposes a `verify()` function:

```
interface IRMNRemote {
  /// @notice Verifies signatures of RMN nodes, on dest lane updates as provided in the CommitReport
  /// @param offRampAddress is not inferred by msg.sender, in case the call is made through ARMProxy
  /// @param merkleRoots must be well formed, and is a representation of the CommitReport received from the oracles
  /// @param signatures rmnNodes ECDSA sigs, only r & s, must be sorted in ascending order by signer address
  /// @param rawVs rmnNodes ECDSA sigs, part v bitmap
  /// @dev Will revert if verification fails
  function verify(
    address offRampAddress,
    Internal.MerkleRoot[] memory merkleRoots,
    Signature[] memory signatures,
    uint256 rawVs
  ) external view;
}
```

This `verify()` function is used by the `OffRamp` to validate merkle roots before persisting them:

```
// OffRamp.sol

function commit(...) {
  // ...
  if (commitReport.merkleRoots.length > 0) {
    i_rmnRemote.verify(address(this), commitReport.merkleRoots, commitReport.rmnSignatures, commitReport.rmnRawVs);
  }
  // ...
}
```

Note that RMN signatures are only validated *if* this report contains roots to commit (it might only have price updates).

The `RMNRemote` contract also has the ability to *curse* arbitrary subjects

```
function curse(bytes16 subject) external;
```

In 1.5, cursing was done automatically by RMN nodes when they detected anomalies. in 1.6, only manual curses are possible, where the RMN contract owner explicitly curses a subject.

RMNHome

The `RMNHome` contract exists only to configure and coordinate offchain systems. It has no direct interactions with other contracts. The `RMNHome` contract contains a `VersionedConfig` struct that is consumed by the RMN and CommitDON to ensure all commits and blessings are made based on the latest config. Ex the `RMNRemote` contract contains:

```
// RMNRemote.sol

struct Config {
  bytes32 rmnHomeContractConfigDigest; // Digest of the RMNHome contract config
  Signer[] signers;
  uint64 minSigners;
}
```

Above, the `rmnHomeContractConfigDigest` is a digest of the RMNHome contract's config. When the `RMNHome` config *changes*, the `RMNRemote` contracts on each chain need to be updated to account for the new config. Any commit reports generated with mis-matched config digests will fail to persist on-chain. This helps ensure config changes are atomic and all commitments / blessings are generated on the latest config.

`RMNHome` can support two concurrent configurations - one "live" and one "pending". These are used as part of CCIP blue/green deployments.

FeeQuoter

Multi lane support

As mentioned in the OnRamp section, all fee and chain family specific logic have been transferred from the `OnRamp` to the `FeeQuoter` to make the `OnRamp` fully destination chain family agnostic. Therefore, similarly to the `OnRamp`, all destination chain specific variables are now stored in a `DestChainConfig` struct with the exception of the token transfer fee configs which require a separate nested mapping.

Variable		Per destination chain	Per token
`isEnabled`	Whether a destination chain is enabled	✅	
`maxNumberOfTokensPerMsg`	Maximum number of distinct ERC20 token transferred per message	✅	
`maxDataBytes`	Maximum payload data size in bytes	✅	
`maxPerMsgGasLimit`	Maximum gas limit for messages targeting EVMs	✅	
`destGasOverhead`	Gas charged on top of the gasLimit to cover destination chain costs	✅	
`destGasPerPayloadByte`	Destination chain gas charged for passing each byte of `data` payload to receiver	✅	
`destDataAvailabilityOverheadGas`	Extra data availability gas charged on top of the message, e.g. for OCR	✅	
`destGasPerDataAvailabilityByte`	Amount of gas to charge per byte of message data that needs availability	✅	
`destDataAvailabilityMultiplierBps`	Multiplier for data availability gas, multiples of bps	✅	
`defaultTokenFeeUSDCents`	Default token fee charged per token transfer	✅	
`defaultTokenDestGasOverhead`	Default gas charged to execute the token transfer on the destination chain	✅	
`defaultTokenDestBytesOverhead`	Default extra data availability bytes charged per token transfer	✅	
`defaultTxGasLimit`	Default gas limit for a tx	✅	
`gasMultiplierWeiPerEth`	Multiplier for gas costs, 1e18 based so 11e17 = 10% extra cost	✅	
`networkFeeUSDCents`	Flat network fee to charge for messages, multiples of 0.01 USD	✅	
`gasPriceStalenessThreshold`	The amount of time a gas price can be stale before it is considered invalid (0 means disabled)	✅	
`enforceOutOfOrder`	Whether to enforce the allowOutOfOrderExecution extraArg value to be true	✅	
`chainFamilySelector`	Selector that identifies the destination chain's family. Used to determine the correct validations to perform for the dest chain	✅	
`s_usdPerUnitGasByDestChainSelector`	The gas price per unit of gas for a given destination chain, in USD with 18 decimals	✅	
`minFeeUSDCents`	Minimum fee to charge per token transfer, multiples of 0.01 USD	✅	✅
`maxFeeUSDCents`	Maximum fee to charge per token transfer, multiples of 0.01 USD	✅	✅
`deciBps`	Basis points charged on token transfers, multiples of 0.1bps, or 1e-5	✅	✅
`destGasOverhead`	Gas charged to execute the token transfer on the destination chain	✅	✅
`destBytesOverhead`	Default data availability bytes that are returned from the source pool and sent to the destination pool	✅	✅
`s_usdPerToken`	The price, in USD with 18 decimals, per 1e18 of the smallest token denomination		✅
`s_usdPriceFeedsPerToken`	Stores the price data feed configurations per token.		✅
`s_premiumMultiplierWeiPerEth`	The multiplier for destination chain specific premiums that can be set by the owner or fee admin. This should never be 0 once set, so it can be used as an isEnabled flag		✅

These variables can be updated by the contract owner through the `applyDestChainConfigUpdates()` function:

```
/// @notice Updates destination chains specific configs.
/// @param destChainConfigArgs Array of destination chain specific configs.
function applyDestChainConfigUpdates(DestChainConfigArgs[] memory destChainConfigArgs) external onlyOwner {
  _applyDestChainConfigUpdates(destChainConfigArgs);
}

/// @notice Internal version of applyDestChainConfigUpdates.
function _applyDestChainConfigUpdates(DestChainConfigArgs[] memory destChainConfigArgs) internal {
  for (uint256 i = 0; i < destChainConfigArgs.length; ++i) {
    DestChainConfigArgs memory destChainConfigArg = destChainConfigArgs[i];
    uint64 destChainSelector = destChainConfigArgs[i].destChainSelector;

    if (destChainSelector == 0) {
      revert InvalidDestChainConfig(destChainSelector);
    }

    DestChainConfig storage destChainConfig = s_destChainConfigs[destChainSelector];
    destChainConfig.router = destChainConfigArg.router;

    emit DestChainConfigSet(
      destChainSelector, destChainConfig.sequenceNumber, destChainConfigArg.router, destChainConfig.allowListEnabled
    );
  }
}
```

Keystone Price Reports

In CCIP v1.6 the `FeeQuoter` supports token prices transmissions from both the commit plugin and Keystone*. Therefore, the `FeeQuoter` now conforms to the `IReceiver` interface and implements the `onReport()` function:

```
/// @notice Handles the report containing price feeds and updates the internal price storage
/// @inheritdoc IReceiver
/// @dev This function is called to process incoming price feed data.
/// @param metadata Arbitrary metadata associated with the report (not used in this implementation).
/// @param report Encoded report containing an array of `ReceivedCCIPFeedReport` structs.
function onReport(bytes calldata metadata, bytes calldata report) external {
  (bytes10 workflowName, address workflowOwner, bytes2 reportName) = metadata._extractMetadataInfo();

  _validateReportPermission(msg.sender, workflowOwner, workflowName, reportName);

  ReceivedCCIPFeedReport[] memory feeds = abi.decode(report, (ReceivedCCIPFeedReport[]));

  for (uint256 i = 0; i < feeds.length; ++i) {
    uint8 tokenDecimals = s_usdPriceFeedsPerToken[feeds[i].token].tokenDecimals;
    if (tokenDecimals == 0) {
      revert TokenNotSupported(feeds[i].token);
    }
    // Keystone reports prices in USD with 18 decimals, so we passing it as 18 in the _calculateRebasedValue function
    uint224 rebasedValue = _calculateRebasedValue(18, tokenDecimals, feeds[i].price);

    //if stale update then revert
    if (feeds[i].timestamp < s_usdPerToken[feeds[i].token].timestamp) {
      revert StaleKeystoneUpdate(feeds[i].token, feeds[i].timestamp, s_usdPerToken[feeds[i].token].timestamp);
    }

    s_usdPerToken[feeds[i].token] =
      Internal.TimestampedPackedUint224({value: rebasedValue, timestamp: feeds[i].timestamp});
    emit UsdPerTokenUpdated(feeds[i].token, rebasedValue, feeds[i].timestamp);
  }
}
```

*Keystone is a new form of price reporting and the details of its implementation are not relevant this documentation.

CCIPHome

CapabilityRegistry

In CCIP v1.6, source and destination NOPs are decoupled. This is achieved through the use of a singular home chain contract on Ethereum mainnet called the `CapabilityRegistry` that contains configuration readable by all NOPs (the `CapabilityRegistry` is out of scope for this review). This does force NOPs to support the home chain but is a practical compromise, and it could, in principle, be moved to another non-chain location.

The `CapbilityRegistry` contains explicitly:

- All CCIP DONs and their configuration
- Each DON contains:
  - The list of participating nodes (in particular, their P2P IDs)
  - The capabilities supported by each node, including the capability configurations.

CCIPHome

Each capability can point to a configuration contract that contains capability specific config logic. In the case of CCIP, that contract is the `CCIPHome` contract. This contract is therefore, like the `CapabilityRegistry`, only deployed on the home chain (i.e. Ethereum) and must conform to the `ICapabilityConfiguration`, with 2 functions:

- `beforeCapabilityConfigSet` which is called by the `CapabilityRegistry` upon adding or updated a config for a particular DON through the `updateDON` function
- `getCapabilityConfiguration` which fetches a capability configuration for a particular DON instance

The role of `CCIPHome` is to store and manage the configuration for the CCIP capability.

There are 2 classes of configuration: the chain configuration and the DON configuration. Each chain has a single configuration:

```
/// @notice Chain configuration.
/// Changes to chain configuration are detected out-of-band in plugins and decoded offchain.
struct ChainConfig {
  bytes32[] readers; // The P2P IDs of the readers for the chain. These IDs must be registered in the capabilities registry.
  uint8 fChain; // The fault tolerance parameter of the chain.
  bytes config; // The chain configuration. This is kept intentionally opaque so as to add fields in the future if needed.
}

/// @notice Chain configuration information struct used in applyChainConfigUpdates and getAllChainConfigs.
struct ChainConfigInfo {
  uint64 chainSelector;
  ChainConfig chainConfig;
}
```

And each DON will have up to 4 configurations, 1 for each plugin (commit and exec), one blue and one green:

```
/// @notice OCR3 configuration.
/// Note that FRoleDON >= fChain, since FRoleDON represents the role DON, and fChain represents sub-committees.
/// FRoleDON values are typically identical across multiple OCR3 configs since the chains pertain to one role DON,
/// but FRoleDON values can change across OCR3 configs to indicate role DON splits.
struct OCR3Config {
  Internal.OCRPluginType pluginType; // ──────────╮ The plugin that the configuration is for.
  uint64 chainSelector; //                        | The (remote) chain that the configuration is for.
  uint8 FRoleDON; //                              | The "big F" parameter for the role DON.
  uint64 offchainConfigVersion; // ───────────────╯ The version of the offchain configuration.
  bytes offrampAddress; // The remote chain offramp address.
  OCR3Node[] nodes; // Keys & IDs of nodes part of the role DON
  bytes offchainConfig; // The offchain configuration for the OCR3 protocol. Protobuf encoded.
}

/// @notice OCR3 configuration with metadata, specifically the config count and the config digest.
struct OCR3ConfigWithMeta {
  OCR3Config config; // The OCR3 configuration.
  uint64 configCount; // The config count used to compute the config digest.
  bytes32 configDigest; // The config digest of the OCR3 configuration.
}
```

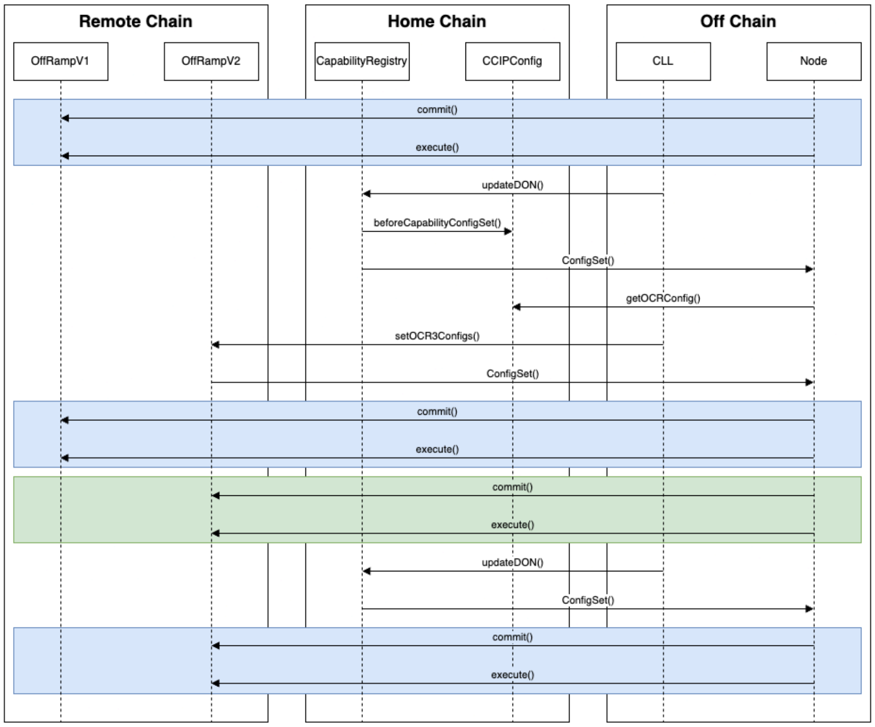
Blue/green

In CCIPv1.6, there is the desire for NOPs to support precisely whichever chains they want to support, but still have interoperability between all chains by default. To accomplish that, there is one OCR instance per destination chain (`OCR3ConfigWithMeta`) with a heterogeneous set of members supporting different chains (including ones who cannot read/write to the destination chain)

All OCR configurations are held on the home chain (`CapabilityRegistry` + `CCIPHome`) which all CCIP NOPs can read from. This way they can join an OCR instance which writes to chains even if they don't support it. Remote chains need to validate signatures on the OCR reports produced, meaning signer changes necessarily require touching both chains.

When performing an upgrade, a blue/green deployment model is used, where 2 versions of CCIP are run in parallel until the upgrade is complete. The active version is referred to as the "blue" version, while the upgraded version is initially set as the "green" version. The "green" version acts as a staging environment to enable an upgrade without downtime and testing. After the upgrade is complete, the "green" (v2) version is promoted to the "blue" version, and the previous "blue" (v1) version is terminated.

Both instances use custom contract transmitters which don't submit to the remote until its digest is configured. It can therefore be rolled back if v2 instance looks unhealthy by setting just the v1 instance (blue) on the home chain. Once the the v2 digest is set on the remote chain, it unblocks the v2 instance and disables the v1 instance. The v1 instance can then be disabled on the home chain.



1. Nodes perform `commit()` and `execute()` operations on the v1 `OffRamp`.
2. `updateDON()` is called on the `CapabilityRegistry` for each destination chain with the updated list of nodes and the corresponding blue and green configs.
3. `beforeCapabilityConfigSet()` is called on the `CCIPHome` contract, which adds the green config for each plugin.
4. Nodes pick up the `ConfigSet` event from the `CapabilityRegistry` and call `getOCRConfig()` on the `CCIPHome` contract to pull the green config + digest, which is used to map the v1.6 `OffRamp` of the destination chain.
5. The `setOCR3Configs()` function is called on the v2 (green) configuration of the v1.6 OffRamp.
6. The `ConfigSet` event is picked by the nodes
7. At this point both blue and green config are operational and nodes can perform `commit()` and `execute()` on both the v1 and v2 `OffRamp`
8. `updateDON` is called on the `CapabilityRegistry` for each destination chain with only the green configs
9. The green config now becomes the new blue config

Non EVM Chains

FeeQuoter

The following struct changes have been implemented for any chain support:

- A `familyTag` variable has been added to the `DestChainDynamicConfig` struct to identify the destination chain's family. It is used to determine the correct validations to perform for the dest chain. For EVM chains the used family tag can be found in the `Internal` library:

```
// bytes4(keccak256("CCIP ChainFamilySelector EVM"))
bytes4 public constant CHAIN_FAMILY_SELECTOR_EVM = 0x2812d52c;
```

- An `EVM2AnyRampMessage` struct has been introduced, replacing the `EVM2EVMMessage` struct for new v1.6 lanes, where all destination specific `address` type variables have been updated to `bytes` :

```
/// @notice Family-agnostic header for OnRamp & OffRamp messages.
/// The messageId is not expected to match hash(message), since it may originate from another ramp family
struct RampMessageHeader {
  bytes32 messageId; // Unique identifier for the message, generated with the source chain's encoding scheme (i.e. not necessarily abi.encoded)
  uint64 sourceChainSelector; // ──╮ the chain selector of the source chain, note: not chainId
  uint64 destChainSelector; //     | the chain selector of the destination chain, note: not chainId
  uint64 sequenceNumber; //        │ sequence number, not unique across lanes
  uint64 nonce; // ────────────────╯ nonce for this lane for this sender, not unique across senders/lanes
}

struct EVM2AnyTokenTransfer {
  // The source pool EVM address. This value is trusted as it was obtained through the onRamp. It can be
  // relied upon by the destination pool to validate the source pool.
  address sourcePoolAddress;
  // The EVM address of the destination token
  // This value is UNTRUSTED as any pool owner can return whatever value they want.
  bytes destTokenAddress;
  // Optional pool data to be transferred to the destination chain. Be default this is capped at
  // CCIP_LOCK_OR_BURN_V1_RET_BYTES bytes. If more data is required, the TokenTransferFeeConfig.destBytesOverhead
  // has to be set for the specific token.
  bytes extraData;
  uint256 amount; // Amount of tokens.
  // Destination chain specific execution data encoded in bytes
  // for an EVM destination, it consists of the amount of gas available for the releaseOrMint
  // and transfer calls made by the offRamp
  bytes destExecData;
}

/// @notice Family-agnostic message emitted from the OnRamp
/// Note: hash(Any2EVMRampMessage) != hash(EVM2AnyRampMessage) due to encoding & parameter differences
/// messageId = hash(EVM2AnyRampMessage) using the source EVM chain's encoding format
struct EVM2AnyRampMessage {
  RampMessageHeader header; // Message header
  address sender; // sender address on the source chain
  bytes data; // arbitrary data payload supplied by the message sender
  bytes receiver; // receiver address on the destination chain
  bytes extraArgs; // destination-chain specific extra args, such as the gasLimit for EVM chains
  address feeToken; // fee token
  uint256 feeTokenAmount; // fee token amount
  uint256 feeValueJuels; // fee amount in Juels
  EVM2AnyTokenTransfer[] tokenAmounts; // array of tokens and amounts to transfer

```

As mentioned in the goals section, there is still a desire to perform destination chain-specific validations, therefore the following logic changes have been implemented:

- Family specific address validation logic has been implemented: if the destination chain is an EVM chain then use the internal `_validateEVMAddress` function, otherwise new internal functions will likely be introduced to perform different sanity checks.

```
/// @notice Validates that the destAddress matches the expected format of the family.
/// @param chainFamilySelector Tag to identify the target family.
/// @param destAddress Dest address to validate.
/// @dev precondition - assumes the family tag is correct and validated.
function _validateDestFamilyAddress(bytes4 chainFamilySelector, bytes memory destAddress) internal pure {
  if (chainFamilySelector == Internal.CHAIN_FAMILY_SELECTOR_EVM) {
    Internal._validateEVMAddress(destAddress);
  }
}

/// @notice This methods provides validation for parsing abi encoded addresses by ensuring the
/// address is within the EVM address space. If it isn't it will revert with an InvalidEVMAddress error, which
/// we can catch and handle more gracefully than a revert from abi.decode.
/// @return The address if it is valid, the function will revert otherwise.
function _validateEVMAddress(bytes memory encodedAddress) internal pure returns (address) {
  if (encodedAddress.length != 32) revert InvalidEVMAddress(encodedAddress);
  uint256 encodedAddressUint = abi.decode(encodedAddress, (uint256));
  if (encodedAddressUint > type(uint160).max || encodedAddressUint < PRECOMPILE_SPACE) {
    revert InvalidEVMAddress(encodedAddress);
  }
  return address(uint160(encodedAddressUint));
}
```

- Custom `extraAgrs` parsing logic: since the `extraArgs` content might change depending on the destination chain, decoding must be customized. For example, a decoded Solana `extraArgs` might look something like:

```
struct SolExtraArgsV1 {
  bool allowOutOfOrder; // Required true initially (possibly forever, seems)
  uint64 computeUnits; // 200k default, 1.4M max https://solana.com/docs/core/fees#compute-unit-limit-1
  uint64 computeUnitPrice; // Optional set gas price https://solana.com/docs/core/fees#prioritization-fees-1
    SolTokenAccountMeta[] tokenAccountMetas;
}
```

  To map the `extraArgs` to its chain family specific decoding the chain family selector is prepended to the encoded bytes. The parsing can then be performed in the `_parseEVMExtraArgsFromBytes()` function:

```
/// @dev Convert the extra args bytes into a struct with validations against the dest chain config.
/// @param extraArgs The extra args bytes.
/// @param destChainConfig Dest chain config to validate against.
/// @return evmExtraArgs The EVMExtraArgs struct (latest version).
function _parseEVMExtraArgsFromBytes(
  bytes calldata extraArgs,
  DestChainConfig memory destChainConfig
) internal pure returns (Client.EVMExtraArgsV2 memory) {
  Client.EVMExtraArgsV2 memory evmExtraArgs =
    _parseUnvalidatedEVMExtraArgsFromBytes(extraArgs, destChainConfig.defaultTxGasLimit);

  if (evmExtraArgs.gasLimit > uint256(destChainConfig.maxPerMsgGasLimit)) revert MessageGasLimitTooHigh();
  if (destChainConfig.enforceOutOfOrder && !evmExtraArgs.allowOutOfOrderExecution) {
    revert ExtraArgOutOfOrderExecutionMustBeTrue();
  }

  return evmExtraArgs;
}

/// @dev Convert the extra args bytes into a struct.
/// @param extraArgs The extra args bytes.
/// @param defaultTxGasLimit default tx gas limit to use in the absence of extra args.
/// @return EVMExtraArgs the extra args struct (latest version)
```

```
function _parseUnvalidatedEVMExtraArgsFromBytes(
  bytes calldata extraArgs,
  uint64 defaultTxGasLimit
) private pure returns (Client.EVMExtraArgsV2 memory) {
  if (extraArgs.length == 0) {
    // If extra args are empty, generate default values
    return Client.EVMExtraArgsV2({gasLimit: defaultTxGasLimit, allowOutOfOrderExecution: false});
  }

  bytes4 extraArgsTag = bytes4(extraArgs);
  bytes memory argsData = extraArgs[4:];

  if (extraArgsTag == Client.EVM_EXTRA_ARGS_V2_TAG) {
    return abi.decode(argsData, (Client.EVMExtraArgsV2));
  } else if (extraArgsTag == Client.EVM_EXTRA_ARGS_V1_TAG) {
    // EVMExtraArgsV1 originally included a second boolean (strict) field which has been deprecated.
    // Clients may still include it but it will be ignored.
    return Client.EVMExtraArgsV2({gasLimit: abi.decode(argsData, (uint256)), allowOutOfOrderExecution: false});
  }

  revert InvalidExtraArgsTag();
}
```

OffRamp

The following changes have been implemented for any chain support:

- All source chain specific `address` types have been updated to `bytes`
- The `messageId` has been decoupled from `hash(message)` as non EVM chain messages might contain extra fields, different encoding schemes and different hashing algorithms, for example:

  Solana OnRamp:

  - `messageId = hash(Sol2AnyMessage msg)`
  - (Sol2Any message could contain some extra fields that are SOL-specific)

  Off-chain:

  - The following conversion is then performed off-chain: `Sol2AnyMessage` -> `Any2AnyMessage` -> `Any2EVMMessage`

  Ethereum `OffRamp`:

  - `committedRootLeafHash = hash(Any2EVMMessage msg)`
  - (Any2EVMMessage will contain some EVM-specific fields, but not the SOL specific fields)

  `hash(Sol2AnyMessage) != hash(Any2EVMMessage)`
- `metadataHash` has been converted to an implicit hash composed of prefix, source chain selector, destination chain selector and onramp address

```
// Hashing all of the message fields ensures that the message being executed is correct and not tampered with.
// Including the known OnRamp ensures that the message originates from the correct on ramp version
hashedLeaves[i] = Internal._hash(
  message,
  keccak256(
    abi.encode(
      Internal.ANY_2_EVM_MESSAGE_HASH,
      message.header.sourceChainSelector,
      message.header.destChainSelector,
      keccak256(onRamp)
    )
  )
);

/// @dev Used to hash messages for multi-lane family-agnostic OffRamps.
/// OnRamp hash(EVM2AnyMessage) != Any2EVMRampMessage.messageId
/// OnRamp hash(EVM2AnyMessage) != OffRamp hash(Any2EVMRampMessage)
/// @param original OffRamp message to hash
/// @param metadataHash Hash preimage to ensure global uniqueness
/// @return hashedMessage hashed message as a keccak256
function _hash(Any2EVMRampMessage memory original, bytes32 metadataHash) internal pure returns (bytes32) {
  // Fixed-size message fields are included in nested hash to reduce stack pressure.
  // This hashing scheme is also used by RMN. If changing it, please notify the RMN maintainers.
  return keccak256(
    abi.encode(
      MerkleMultiProof.LEAF_DOMAIN_SEPARATOR,
      metadataHash,
      keccak256(
        abi.encode(
          original.header.messageId,
          original.receiver,
          original.header.sequenceNumber,
          original.gasLimit,
          original.header.nonce
        )
      ),
      keccak256(original.sender),
      keccak256(original.data),
      keccak256(abi.encode(original.tokenAmounts))
    )
  );
}
```

- Similarly as in the `FeeQuoter`, `EVM2EVMMessage` structs have been replaced by the `Any2EVMRampMessage` struct.

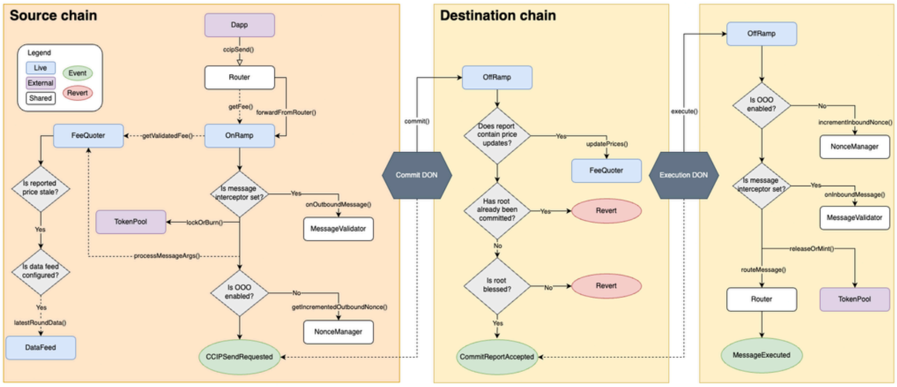
```
/// @notice Family-agnostic header for OnRamp & OffRamp messages.
/// The messageId is not expected to match hash(message), since it may originate from another ramp family
struct RampMessageHeader {
  bytes32 messageId; // Unique identifier for the message, generated with the source chain's encoding scheme (i.e. not necessarily abi.encoded)
  uint64 sourceChainSelector; // ──╮ the chain selector of the source chain, note: not chainId
  uint64 destChainSelector; //     │ the chain selector of the destination chain, note: not chainId
  uint64 sequenceNumber; //        │ sequence number, not unique across lanes
  uint64 nonce; // ────────────────╯ nonce for this lane for this sender, not unique across senders/lanes
}

struct Any2EVMTokenTransfer {
  // The source pool EVM address abi.encoded to bytes. This value is trusted as it is obtained through the onRamp. It can be
  // relied upon by the destination pool to validate the source pool.
  bytes sourcePoolAddress;
  address destTokenAddress; // ──╮ Address of destination token
  uint32 destGasAmount; //───────╯ The amount of gas available for the releaseOrMint and transfer calls on the offRamp.
  // Optional pool data to be transferred to the destination chain. Be default this is capped at
  // CCIP_LOCK_OR_BURN_V1_RET_BYTES bytes. If more data is required, the TokenTransferFeeConfig.destBytesOverhead
  // has to be set for the specific token.
  bytes extraData;
  uint256 amount; // Amount of tokens.
}

/// @notice Family-agnostic message routed to an OffRamp
/// Note: hash(Any2EVMRampMessage) != hash(EVM2AnyRampMessage), hash(Any2EVMRampMessage) != messageId
/// due to encoding & parameter differences
struct Any2EVMRampMessage {
  RampMessageHeader header; // Message header
  bytes sender; // sender address on the source chain
  bytes data; // arbitrary data payload supplied by the message sender
  address receiver; // receiver address on the destination chain
  uint256 gasLimit; // user supplied maximum gas amount available for dest chain execution
  Any2EVMTokenTransfer[] tokenAmounts; // array of tokens and amounts to transfer
}
```

One thing to note is that the `OnRamp` will emit a `EVM2AnyRampMessage` in the `CCIPSendRequested` event while the `OffRamp` `execute()` function will receive `Any2EVMRampMessage` as arguments in its report. The conversion from `EVM2AnyRampMessage` to `Any2EVMRampMessage` is performed off-chain, where the plugins will be in charge of decoding the `extraArgs` from the `EVM2AnyRampMessage` and constructing the `Any2EVMRampMessage`.

Call Flow



1. Dapp or user calls `ccipSend()` on the `Router`.
2. `Router` calls `getFee()` on the `OnRamp` to compute the fee.
3. CCIP v1.6 now outsources the fee specific logic to the `FeeQuoter` so `getFee()` calls `getValidatedFee()` on the `FeeQuoter` which also performs some message validation.
4. `FeeQuoter` will check for reported prices (prices transmitted through the commit plugin or Keystone) staleness and fallback on configured price feeds if stale.
5. `Router.ccipSend()` logic resumes and then calls `forwardFromRouter` on the `OnRamp`.
6. If a message interceptor is set, which for v1.6 will be the aggregate rate limiter, then the hook `onOutboundMessage` function is called on it.
7. `lockOrBurn` is called on the `TokenPool`.
8. The `processMessageArgs()` function is then called on the `FeeQuoter` to convert the message fee to juels, extract extra args variables and perform pool return data validations.
9. If OOO execution is disabled the sender's outbound nonce is incremented on the `NonceManager`.
10. Once the commit DON picks up the `CCIPSendRequested` event it calls `commit()` on the `OffRamp`.
11. If the commit report contains price updated `updatePrices()` is called on the `FeeQuoter`.
12. If the root has already been committed then the transaction reverts.
13. The `OnRamp` then calls `isBlessed()` on the `RMN`. If a root is not blessed then the call reverts, else the root is blessed and the report is committed.

14. Once the execution DON picks up the `CommitReportAccepted` event it calls `execute()` on the `OffRamp`.
15. If a message interceptor is set, which again for v1.6 will be the aggregate rate limiter, then the hook `onInboundMessage` function is called on it.
16. If OOO execution is disabled the sender inbound nonce in incremented on the `NonceManager`.
17. `releaseOrMint()` is called on the `TokenPool`.
18. Finally message is routed to the `Router` for receiver call.

Special Areas of Concern

These are the areas where the auditors should pay special attention:

- DoS: for example a condition that would block all lanes because of a single lane issue
- Security check bypass due to the new multi lane nature of the contracts: for example using the data from one lane to execute a message on another lane
- Things that might have been missed on the chain agnostic side: for example a non EVM chain integration that would somehow break the current security model
- Hashing collisions
- Replay attacks messages/price reports (same chain/cross chain)
- Upgrades leading to unintentional side effects
- Nonce ordering not honoured
- Nonce ordering leading to stuck messages
- Multi-source batch getting stuck (one source blocking another)
- Unintentional interactions between 1.5 self-serve pools and V1.6 multi-ramps
- Over/under payment / billing bugs
- Wasting NOP gas
- Unrecoverable states in the CCIPHome / RMNHome state transitions

Known / Out of Scope Issues:

Token developer is malicious

- Any malicious token developer can negatively impact transactions that contain their token.
- Any malicious token developer can make transactions that contain their token fail forever, leading to the loss of any funds in that transaction.
- Users should be cautious about sending tokens they do not trust.
- Sending a single token per transaction will prevent any contagion to other tokens.
- On some chains, there are methods through which a malicious token developer can cause an entire transaction to revert without being caught by the try-catch, such as overflowing ZK proof size on EVM-compatible zkRollups. In such edge cases, a malicious token developer can block following transactions from the same sender at the destination when not using out-of-order execution, potentially causing unbounded loss.
- Anyone can add any token with any name or affiliation. The CCIP Owner has zero control over what is listed and cannot modify or remove any tokens. This means there could be tokens listed with names or affiliations that are not endorsed by the CCIP Owner.
- Malicious tokens or token pools will be able to re-enter.
  - This should not have any negative impact on the CCIP system.
  - This does allow them to re-order some events, but never to double spend.

Token developer responsibility

- When a token developer misconfigures their own tokens/pools, users could potentially not receive these tokens.
- The CCIP Owner has no special access to resolve misconfigurations, only the token developer can do so.
- Token developers can deploy token pools without rate limits, which is different from today where every token pool is rate-limited.

Compatibility

- Some tokens will not work.
- For example, fee-on-transfer tokens won't work because they require multiple hops in the CCIP contracts.
- They will still be able to register their token, as this process is fully permissionless. This could negatively impact transactions containing these tokens.
- Pool `releaseOrMint` is bound by a per-ramp `maxPoolReleaseOrMintGas`, and Token transfer is bound by a per-ramp `maxTokenTransferGas`. As a result, pools or tokens that are extremely gas-intensive may not be supported. Similarly, pool data that can be relayed from source to destination is bound as well.
- Not every token will be able to permissionlessly register such as tokens that do not expose the `owner` and don't have the `getCCIPAdmin` function.
- These can still be onboarded in the same way all current tokens are onboarded: manually through the CCIP Owner.

Other

- CCIP Owner is a trusted role.
- The multi-signature contract structure includes the entire DON and therefore inherits the security of the entire DON.
- Solidity 0.8.24 is used and CCIP will be deployed on various chains that don't support the newer Solidity features.
  - The Paris hard fork is explicitly compiled to ensure compatibility.
- The aggregate rate limiter only works for tokens that have prices, as it's denominated in USD.
- Self-serve tokens will not have prices, as it would be impractical to acquire accurate price sources for any arbitrary token, and writing these prices onchain would be economically unsustainable, and easily exploitable.
- This means that tokens default to *not* be included in the aggregate rate limiter.
- Router API `getSupportedTokens()` is now deprecated, calling it will revert.

- There is no longer an official API that returns all supported tokens for a given destination in 1 call.
- One should iterate through tokens via `getAllConfiguredTokens()` in TokenAdminRegistry and then call `isSupportedChain()` on its token pool.