Payment Abstraction v0.1

Motivation

Payment Abstraction is a system of onchain smart contracts that aim to reduce payment friction for Chainlink services. The system is designed to (1) accept fees in various tokens across multiple blockchain networks, (2) consolidate fee tokens onto a single blockchain network via Chainlink CCIP, (3) convert fee tokens into LINK via Chainlink Automation, Price Feeds, and existing Automated Market Maker (AMM) Decentralized Exchange (DEX) contracts, and (4) pass converted LINK into a dedicated contract for withdrawal by Chainlink Network service providers.

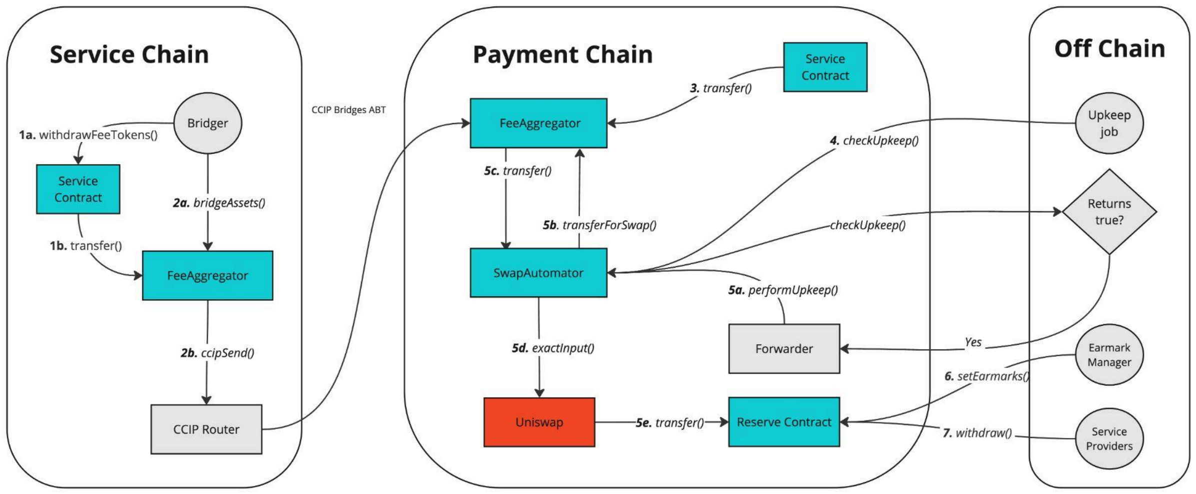
Glossary

Term	Definition
Service provider	An onchain address that is owed LINK payments for providing services
Earmark	Information describing any payments owed to a Service Provider and data around those payments
LINK	The native token of the Chainlink ecosystem. LINK is an ERC677 token that inherits functionality from the ERC20 token standard and allows token transfers to contain a data payload.

Juels	The smallest denomination of LINK. 1,000,000,000,000,000,000 (1e18) Juels are equal to 1 LINK. This is similar to Wei, which is the smallest denomination of ETH.
ABT (Accepted Billing Token)	Onchain tokens that have been allowlisted to be accepted as a form of fee payment within the system
CCIP (Cross-Chain Interoperability Protocol)	A secure cross-chain messaging protocol for transferring data and/or tokens between blockchain networks, secured by Chainlink decentralized oracle networks
CLA (Chainlink Automation) job	A decentralized automation service, where onchain transactions are executed based on time or event triggers, secured by Chainlink decentralized oracle networks
DON (Decentralized Oracle Network)	A decentralized network of independent node operators that achieves consensus on a specific computation and/or data point. DONs are used in the Chainlink ecosystem to secure services such as CCIP and Data Feeds.
Payment chain	The blockchain network where fees are aggregated from all other service chains before being swapped to LINK. The payment chain is also where LINK fees are made available to service providers. There is one payment chain. Throughout this

	document, the payment chain is always Ethereum.
Service chain	A blockchain network where fees from Chainlink services are collected before they are bridged to the Payment Chain. There can be multiple Service Chains.

Design



1. The Service Contracts accrue fees. The fees can be permissionlessly pushed to the FeeAggregator contract.
2. The Bridger manually bridges assets from the service chain FeeAggregator to the payment chain FeeAggregator using CCIP.
    a. This is done by calling the bridgeAssets function on the FeeAggregator.
    b. The FeeAggregator will interact with the CCIP router to bridge assets to the payment chain
3. Service Contracts accrue fees on the payment chain, which can be permissionlessly pushed to the FeeAggregator.
4. Chainlink Automation will periodically call the SwapAutomator contract's checkUpkeep function offchain. The function determines the list of assets & amounts to swap.
5. If the checkUpkeep call determines that at least one swap should occur:
    a. the SwapAutomator's performUpkeep function is called by the nodes in the Automation DON to perform the swaps.
    b. The SwapAutomator contract calls the FeeAggregator transferForSwap function.
    c. The FeeAggregator transfers the assets to swap to the SwapAutomator.
    d. The SwapAutomator swaps the assets on Uniswap and sets the recipient address to the Reserves contract.
    e. Uniswap sends the swapped LINK to the Reserves contract.
6. The earmark managers update the owed LINK on the Reserves contract
7. Service providers draw down the owed LINK from the Reserves contract

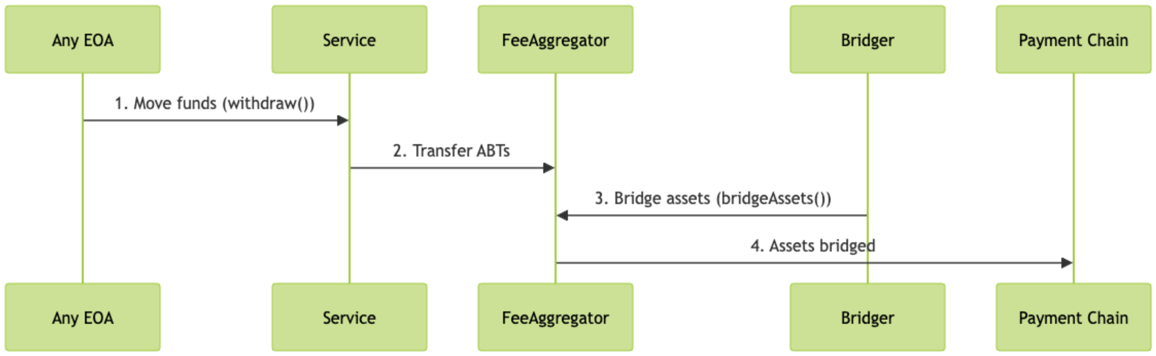
Components

Component	Responsibility
FeeAggregator	ABT payments collected from Service contracts will accumulate in this smart contract. The contract is capable of both bridging and receiving ABTs on a configuration basis. The ABTs will continue to accumulate until either they are bridged, or swapped to LINK by the `Swap CLA`.
FeeRouter	Optional smart contract that can be deployed & placed between the Service contract & the FeeAggregator. The contract prevents automatic flow of fees into the payment abstraction system on local chains.
SwapAutomator	The SwapAutomator smart contract is a CLA Upkeep implementation. It triggers swaps through Uniswap V3 if certain conditions are met. The contract is parameterized to give control on how often a swap should happen, the tolerable slippage parameters,

	and the Uniswap V3 routes to use for the swaps.
Swap CLA	A registered Chainlink Automation upkeep that automates the SwapAutomator contract's `performUpkeep` executions through the custom logic trigger (`checkUpkeep`)
Reserves Contract	LINK received from bridging ABTs from the service chain and swapped on the payment chain will accumulate in the `Reserves` contract on the payment chain. An address granted the `EARMARK_MANAGER_ROLE` can update earmarks to specify how much LINK each service provider is owed.

Flows

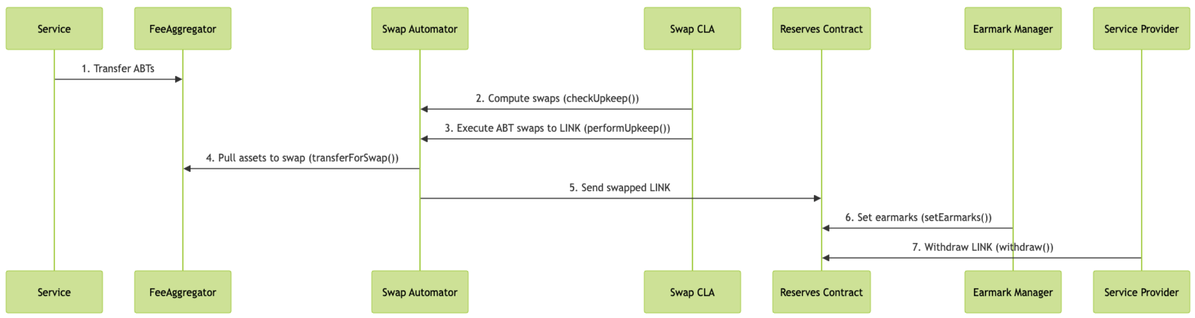
Service Chain



1. Chainlink services bill users internally and accumulate fees on their contracts

2. Any Externally Owned Account (EOA) can permissionlessly transfer the ABTs to the FeeAggregator
3. After being initiated by the bridger, the FeeAggregator bridges the assets to another FeeAggregator contract on the payment chain.
    a. **Note:** bridging may also be done to other service chains instead of the payment chain

Payment Chain



1. Chainlink services bill users internally and accumulate fees on their contracts
2. The `Swap CLA` job monitors the balance of the `FeeAggregatorReceiver` and periodically swaps ABTs to LINK, using the SwapAutomator
    a. **Note:** a SwapAutomator may also be attached to a FeeAggregator on a service chain
3. The swapped & received LINK is sent to the `Reserve` contract
4. Earmark managers write earmarks to the `Reserve` contract to specify how much LINK `Service Providers` are owed.
5. LINK is withdrawn by the `Service Providers`

Roles

Role	Description	Callable Functions
`DEFAULT_ADMIN`	Owner of the contracts. Manages roles,	- `grantRole`

	configurations, and can emergency withdraw assets.	- `beginDefaultAdminTransfer` - `emergencyWithdraw` Configs: - `setDestinationReceiver` - `applyAllowlistedSendersUpdates` - `applyAllowlistedReceiversUpdates` - `setLINKReceiver`
`EARMARK_MANAGER_ROLE`	Manages updating Earmarks and the amounts owed for each Service Provider as well as the allowlisted Service Provider list.	- `setEarmarks` - `addAllowlistedServiceProviders` - `removeAllowlistedServiceProviders`
`PAUSER_ROLE`	Pauses the contract when an emergency is detected.	- `emergencyPause`
`UNPAUSER_ROLE`	Unpauses the contract when an emergency is resolved. This is separate from the `PAUSER_ROLE` to decouple the role.	- `emergencyUnpause`
`BRIDGER_ROLE`	Bridges allowlisted assets on the FeeAggregator contracts across chains.	- `bridgeAssets` - `transferAllowlistedAssets`

SWAPPER_ROLE	Assigned to SwapAutomator contracts, this role allows for pulling allowlisted assets from FeeAggregator contracts to swap.	- transferForSwap
WITHDRAWER_ROLE	Withdraws non-allowlisted assets from the FeeAggregator contracts..	- withdrawNonAllowlistedAssets
ASSET_ADMIN_ROLE	Manages the list of allowlisted assets on the FeeAggregator contracts. Manages the swap parameters on the SwapAutomator contracts	- applyAllowlistedAssetsUpdates - applyAssetSwapParamsUpdates - setDeadlineDelay

Mechanisms

Bridging Assets

The FeeAggregator contract is dual purpose, it can act as both a receiver and sender of assets.

Sending

The Bridgers will bridge assets on a service chain by calling the bridgeAssets function.

```javascript
/// @notice Bridges assets from the source chain to a receiving
/// address on the destination chain
/// @param bridgeAssetAmounts The amount of assets to bridge
/// address on the destination chain
/// @return bytes32 The bridging message ID
function bridgeAssets(
  Client.EVMTokenAmount[] calldata bridgeAssetAmounts,
  uint64 destinationChainSelector,
  bytes calldata bridgeReceiver,
  bytes calldata extraArgs,
) external returns (bytes32);
```

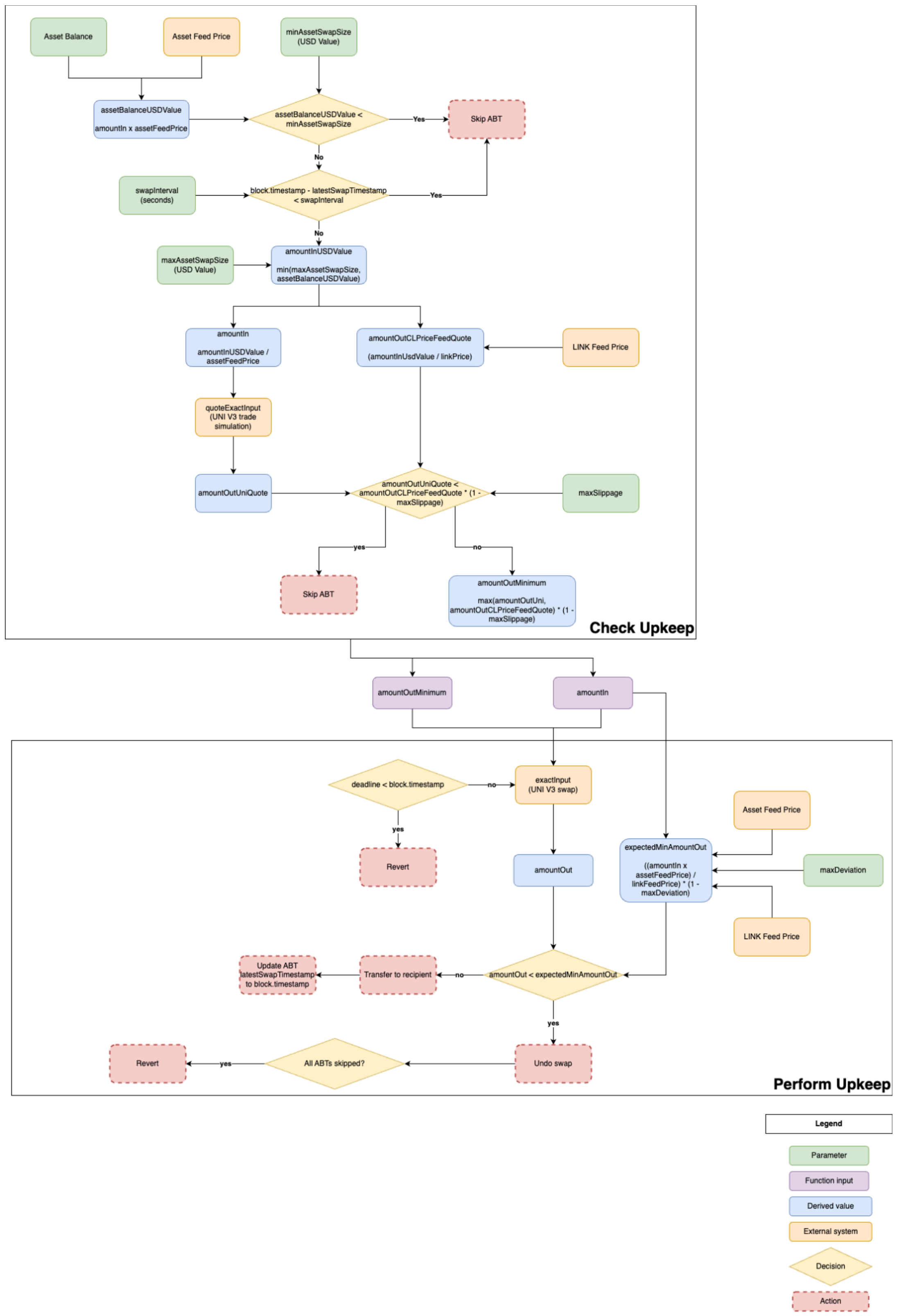
At a high level this function will do the following:

- Checks that the msg.sender has the BRIDGER_ROLE.
- Verify that the destinationChainSelector and bridgeReceiver are allowlisted.
- Construct a Client.EVM2AnyMessage to bridge tokens to the payment chain.
- Calculate the fee required to bridge tokens in juels.
- Approve the CCIP RouterClient contract to spend the required fees.
- Call ccipSend on the RouterClient to send the message to the destination chain.

Receiving

When acting as a receiver, the FeeAggregator does not implement any ccipReceive callback. Instead, it only receives tokens.

Swapping Strategy


Asset Balance
Asset Feed Price
minAssetSwapSize (USD Value)
assetBalanceUSDValue
amountIn x assetFeedPrice
assetBalanceUSDValue < minAssetSwapSize
Yes
Skip ABT
No
swapInterval (seconds)
block.timestamp - latestSwapTimestamp < swapInterval
Yes
No
maxAssetSwapSize (USD Value)
amountInUSDValue
min(maxAssetSwapSize, assetBalanceUSDValue)
amountIn
amountInUSDValue / assetFeedPrice
amountOutCLPriceFeedQuote
(amountInUsdValue / linkPrice)
LINK Feed Price
quoteExactInput (UNI V3 trade simulation)
amountOutUniQuote
amountOutUniQuote < amountOutCLPriceFeedQuote * (1 - maxSlippage)
maxSlippage
yes
no
Skip ABT
amountOutMinimum
max(amountOutUni, amountOutCLPriceFeedQuote) * (1 - maxSlippage)
Check Upkeep
amountOutMinimum
amountIn
deadline < block.timestamp
no
exactInput (UNI V3 swap)
yes
Revert
amountOut
expectedMinAmountOut
((amountIn x assetFeedPrice) / linkFeedPrice) * (1 - maxDeviation)
Asset Feed Price
maxDeviation
LINK Feed Price
Update ABT latestSwapTimestamp to block.timestamp
Transfer to recipient
no
amountOut < expectedMinAmountOut
yes
Revert
yes
All ABTs skipped?
Undo swap
Perform Upkeep
Legend
Parameter
Function input
Derived value
External system
Decision
Action

- Non-LINK ABTs need to be swapped to LINK automatically and must be executed in batches with specified amounts in order to minimize the loss due to slippage.
- Since the swaps will be done on Uniswap V3, the transaction can be exposed to front running risks such as sandwich attacks. To mitigate this risk, we will use the Uniswap's amountOutMinimum parameter which will be computed using Chainlink Data Feeds. Using prices directly from the Uniswap pools can be dangerous since pool balances can also be manipulated. Uniswap's swap function will revert if the amountOutMinimum from the swap is less than the value passed in.
- ABTs swaps are skipped if the Uniswap quote is significantly worse than the Data Feed quote, OR if the asset balance is below the min swap size in USD
- **Transaction Delays:** There can be delays between checkUpkeep results and performUpkeep due to latency with offchain CLA infrastructure or onchain transaction processing
- **Simulated reverts:** performUpkeep is simulated within CLA offchain - if the simulation fails, the onchain transmit will not be executed.
- **Onchain reverts** can occur due to:
    - Deadline condition not being met due to transaction delays
    - ABT swap failures, which can be caused due to:
        - MEV
        - Pool activity
        - Data Feed price being significantly better than the actual Uniswap quote (e.g., due to latency delays)

Swap Parameters

- The swap parameters will be stored in the SwapAutomator contract.
- Swap parameters can only be set for assets found on the allowlist of the FeeAggregator contract.
- Setting of swap parameters of non-allowlisted assets will result in a transaction revert.
- Swap parameters can be added/updated or removed by calling the *applyAssetSwapParamsUpdate*

```JavaScript
/// @notice The maximum amount of seconds the swap transaction is valid for
```

```
uint96 private s_deadlineDelay;


/// @notice Mapping of assets to their swap parameters
mapping(address asset => SwapParams swapParams) private s_assetSwapParams;

/// @notice The parameters to perform a swap
struct SwapParams {
    AggregatorV3Interface oracle; // ─╮ The asset usd oracle
    uint16 maxSlippage; //            │ The maximum allowed slippage for the
swap in basis points
    uint16 maxPriceDeviation; //      │  The maximum allowed one-side deviation
of actual swapped out amount
    //                                │  vs CL oracle price feed estimated
amount
    uint64 swapInterval; // ──────────╯ The minimum interval between swaps
    uint128 minSwapSizeUsd; // ───────╮ The minimum swap size expressed in USD 8
decimals
    uint128 maxSwapSizeUsd; // ───────╯ The maximum swap size expressed in USD 8
decimals
    bytes path; //                      The swap path
}
```

Example Swap Condition

- As an example, let's assume there is 100,000 USDC in the FeeAggregator contract
- Let's assume the following exchange rates for simplicity
    - 1 USDC = 1 USD
    - 1 LINK = 15 USD
- Let's assume the following configured parameters for swapping USDC
    - maxSlippage = 0.01 (This is estimated offchain and represents the maximum percentage we expect to lose.)
    - maxSwapAmountUSD = 50,000 USD
    - minSwapAmountUSD = 30,000 USD
- We need to swap some of the 100,000 USD as it is greater than the minSwapAmountUSD of 30,000 USD

```
Unset
assetUSDPrice = 1 USD
faBalance = 100,000 USDC / assetUSDPrice
swapAmount = Math.min(maxSwapAmountUSD, faBalance)
           = Math.min(50,000, 100,000)
           = 50,000 USDC

feedQuote = swapAmount * assetUSDPrice / linkUSDPrice = 50,000 *
1 / 15 = 3,333.333 LINK

swapPath = USDC-0.05%->WETH-0.3%->LINK
simulatedUNIQuote = quoteExactInput(swapPath, swapAmount) = 3320
LINK

minLINKReceivedFromSwap = max(simulatedUNIQuote,feedQuote) * (1 -
maxSlippage) = max(3320, 3333.333) * (1 - 0.01) = 3300

Therefore a swap should only succeed if the system will receive
at least 3,300
LINK from swapping 50,000 USDC.  Receiving less than 3,300 LINK
should cause the swap to revert.
```

Swap Automation

Swapping is achieved by using a Chainlink Automation job that executes the swaps when the swapping conditions have been met. The currently implemented DEX to perform the swaps is Uniswap V3 (this may change in the future, but that is outside the scope of this current design).

The call stack is as follows:

1. Chainlink Automation invokes checkUpkeep & executes performUpkeep on the SwapAutomator with the resulting data
2. SwapAutomator contract calls the FeeAggregator's transferForSwap function to pull tokens to swap
3. FeeAggregator transfers the amount of assets to the SwapAutomator
4. The SwapAutomator will swap the assets using Uniswap and set the recipient address to the Reserve's contract address

The SwapAutomator needs to inherit from the AutomationCompatible interface and implement the checkUpkeep and performUpkeep functions.

transferForSwap (FeeAggregator)

The FeeAggregator will have a `transferForSwap` function that is only callable by the contracts with the `SWAPPER_ROLE`. In this case, the SwapAutomator contract has been given the role. This function is used by the `SwapAutomator` to pull assets from the `FeeAggregator` before the `SwapAutomator` performs swaps.

```
JavaScript
/// @inheritdoc IFeeAggregator
/// @dev precondition The caller must have the SWAPPER_ROLE
/// @dev precondition The asset must be allowlisted
/// @dev precondition The amount must be greater than 0
function transferForSwap(
  address to,
  address[] calldata assets,
  uint256[] calldata amounts
) external whenNotPaused onlyRole(Roles.SWAPPER_ROLE);
```

checkUpkeep (SwapAutomator)

```
JavaScript
function checkUpkeep(bytes calldata checkData)
    external view override returns (bool upkeepNeeded, bytes
memory performData);
```

The checkUpkeep function is intended to be a read-only function that Automation nodes will read from offchain and will execute onchain if the upkeepNeeded return variable is marked as true. Additionally Automation nodes will pass in the performData to the performUpkeep function when executing the actual job.

At a high level, this function is responsible for iterating through the list of allowlisted assets and constructing an array of Uniswap ExactInputParams that the performUpkeep function will then use to execute swaps.

performUpkeep (SwapAutomator)

```javascript
function performUpkeep(bytes calldata performData) external
override;
```

The performUpkeep function is the function the Automation DON will call to execute transactions on target contracts. This function does incur gas hence should be as lightweight as possible. At a high level, this function will decode performData into an array of ExecuteSwapData elements, pull the assets from the FeeAggregator by calling the `transferForSwap` function and finally perform the swap by calling Uniswap's `exactInput` functions. In addition this function will also be access controlled so that only its Forwarder address can call it in order to ensure that it is only callable by the automation job.

Uniswap V3 Swapping

The system uses simple order routing to allow executing swaps to rely on a fully onchain DEX swapping solution. This is utilized by using the Uniswap V3 exactInput() function which takes the following struct as parameter:

```javascript
struct ExactInputParams {
    bytes path;
    address recipient;
    uint256 amountIn;
    uint256 amountOutMinimum;
}
```

Parameter	Description	State variable name

path	The path to swap tokens on Uniswap.	Set from s_assetSwapParams in checkUpkeep
recipient	The destination address of the outbound asset	Set to s_linkReceiver in checkUpkeep
amountIn	The amount of inbound asset to swap for LINK	Dynamically calculated based on asset balance, capped to max swap size.
amountOutMinimum	The minimum amount of juels to receive from the swap	Dynamically calculated based on feed and simulated Uniswap quotes

Service Providers

Service Providers will be represented by mapping addresses to the `ServiceProvider` struct and will be added to an allowlist as shown below.

Unset
```
struct ServiceProvider {
      int96 linkBalance;
      uint96 earmarkCounter;
}

EnumerableSet.AddressSet private s_allowlistedServiceProviders;
mapping(address serviceProvider => ServiceProvider serviceProviderInfo) private
s_serviceProviders;
```

The `linkBalance` field is declared as an `int96` so that it can support negative numbers in case the `EARMARK_MANAGER` needs to undo an owed payment as explained here.

Earmarks

Earmarks are managed by any address granted the EARMARK_MANAGER_ROLE, which will allow the address to update the Earmark data for a Service Provider and the amount of LINK the Service Provider is owed. In order to add Earmarks to a service provider, they must first be allowlisted.

Owed Payments

The amount of LINK owed to a Service Provider is stored in the `ServiceProvider` struct and can be increased or decreased by the `EARMARK_MANAGER` by calling `setEarmarks.`

```
Unset
struct Earmark {
      address serviceProvider;
      int96 amountLinkOwed;
      bytes data;
}

function setEarmarks(Earmark[] calldata earmarks) external;
```

Individual earmarks are not stored onchain. Instead, they are emitted as an event, and are uniquely identified by the *serviceProvider* address and the *earmarkCounter*

```
Unset
    uint256 earmarkCounter = ++serviceProviderInfo.earmarkCounter;
    serviceProviderInfo.linkBalance += amountLinkOwed;

    s_serviceProviders[serviceProvider] = serviceProviderInfo;
```

```
        emit EarmarkSet(serviceProvider, earmarkCounter, amountLinkOwed,
earmarks[i].data);
```

Example

Time	Action	amountOwed
0	`EARMARK_MANAGER` calls `setEarmark` to increase the `linkBalance` to a Service Provider by 100 LINK	100 LINK
1	Service Provider calls `withdraw` to withdraw owed balance (100 LINK)	0 LINK
2	`EARMARK_MANAGER` calls `setEarmark` to undo setting the `linkBalance` at T0	0 - 100 = -100 LINK
3	Service Provider calls `withdraw` to partially withdraw the remaining 50 LINK. This call will revert as the `linkBalance` to the Service Provider is negative. The `linkBalance` has not changed as there was no transfer of LINK	-100 LINK
4	`EARMARK_MANAGER` calls `setEarmark` to increase the `linkBalance` to a Service Provider by 250 LINK to compensate them for future work.	-100 + 250 = 150 LINK
5	Service Provider calls `withdraw` to withdraw 150 LINK.	0

Earmark Data

The data field in Earmark can be used by the EARMARK_MANAGER to store arbitrary data on the contract. As the data field is just a bytes array, the shape of the struct can be changed in the future and decoded offchain.

```
Unset
event EarmarkSet(
      address serviceProvider,
      uint256 serviceProviderNonce,
      uint256 amountLinkOwed,
      bytes earmarkData
);
```

Reserves Withdrawals

Service providers will be able to withdraw an amount of LINK equal to the amount of LINK they are owed by calling the withdraw function. At a high level the withdraw function will read the amount of LINK a serviceProvider is owed and transfers the linkBalance amount of LINK to the serviceProvider address. It will not impose any restriction on the frequency a service provider can call this function to allow an address to withdraw owed payments at any frequency. The function is public and permissionless. This function will revert if any of the serviceProviders specified has a linkBalance <= 0.

```
Unset
function withdraw(address[] serviceProviders) external;
```

Security Assumptions

Trusted Addresses

All role granted addresses should be assumed to be non-malicious, including Chainlink Automation nodes. Nonetheless, some critical validations on trusted inputs are performed to increase the security and help protect against worst-case scenarios. This is the case in the performUpkeep function which validates some values that are relayed by the Chainlink Automation nodes from the checkUpkeep function.

checkUpkeep	**performUpkeep**	**performUpkeep impact**
Check feed staleness	✅	
Checks asset oracle address != address(0)	❌	Not needed, since it will revert if oracle == address(0)
Checks assetPrice != 0	✅	
Checks asset price feed staleness	✅	
Checks swap interval	❌	Low impact if the trade ends up being executed more frequently than the swap interval. The expected amount is bounded by the maxPriceDeviation parameter.
Bounds asset USD value to minimum	❌	Low impact if the swap ends up being slightly below the minimum USD value. Adding the check would significantly increase gas costs.
Bounds asset USD value to maximum	❌	Low impact if the swap ends up being above the maximum USD value. Adding the check would significantly increase gas costs.
Checks uniswap quote against maxSlippage	✅ (maxPriceDeviation)	
Valid asset check	✅	

Valid LINK Receiver check	✅	

Config Validation

Some state configuration data validations are performed offchain instead of onchain e.g. the asset swap path only has a length validation in applyAssetSwapParamsUpdates but additional format sanity checks are performed offchain.

Expected onchain config validations include:

- Zero value checks
- Min vs max sanity checks (min < max, min > MIN_BOUND, max < MAX_BOUND)
- Removing a non listed value from a list
- Updating a non listed value that should have been listed (e.g. setting swap params for a non allowlisted asset)
- Depending list length matching (e.g. list of asset addresses + list of asset swap params)
- No ops (e.g. updating a value to the same value)

Known Limitations

- **Uniswap swapping strategy**: we acknowledge that the current swapping strategy is not the most efficient one (e.g. we could improve it with smart order routing). The current option to use fixed path swaps instead of smart order routing / DEX Aggregation was an intentional design choice to limit the swap strategies to a permissionless DEX option with minimal off-chain requirements.
- **MEV that results in increased slippage:** we acknowledge that there is an inherent risk of the pools being MEV'ed in such a way that the swaps result in higher slippages than anticipated, which could lead to trade reverts. As long as the successful swaps are below our maximum tolerable slippage, MEV pushing to upper bound should not be considered a loss of funds. (e.g. for 0.5% slippage tolerances, it is tolerable to execute at 0.5% instead of 0.2% in the worst case scenario)
- **Delays between checkUpkeep and performUpkeep execution**: we acknowledge the risk, and will mitigate it by the deadline delay, max deviation and slippage parameters. The contract's performUpkeep execution assumes that the inputs might be stale.
- **No recomputation of checkUpkeep data in performUpkeep:** the performUpkeep function performs some critical checks, such as the check against maxPriceDeviation to protect against swaps that result in significantly lower swap results. However, we chose

not to perform less critical validations (e.g. USD value of an asset being swapped within min-max USD swap size bounds) to save on gas costs & reduce complexity.

- **Malicious token processing:** all tokens will be vetted before onboarding them to the system. Malicious tokens are not a concern. Due to all tokens being reviewed, upgradability / pausability / blocklists / balance modifications / fee token transfers are out of scope.
- **Swapping optimization by utilising intermediate L2 routes:** for the current iteration, swaps will only occur on one network.
- **Gas optimizations for checkUpkeep:** the function is executed off-chain, gas optimizations will not have significant improvements
- **Maximum swap gas controls:** maximum gas thresholds will be controlled via Chainlink Automation's max gas threshold functionality
- **Re-entrancy in the Reserves contract:** the withdraw functionality in the Reserves contract will only allow withdrawing to allowlisted providers, and will use the LINK token, so there are no concerns for re-entrancy due to trust assumptions.
- **Negative balances for earmarks:** In rare circumstances, service providers may have negative balances in the Reserves contract, for scenarios when earmarks need to be retroactively adjusted. We acknowledge the risk of delisting a service provider with negative balances, where we may not resolve the negative LINK balance.
- **Swaps might not be executed due to spread between feed prices & Uniswap prices:** we acknowledge that some swaps might revert due to the differences between Chainlink Data Feed and Uniswap pool prices. This decision was made by design to execute trades against Chainlink Data Feed prices as the source of truth.
- **Exploits of dependencies:** malicious exploits on dependent protocols that are outside of the control of the system, e.g., an exploit in a Uniswap V3 liquidity pool contract.
- **In-depth validations for swap params:** asset swap path & the oracle being a correct Chainlink Data Feed address is validated off-chain