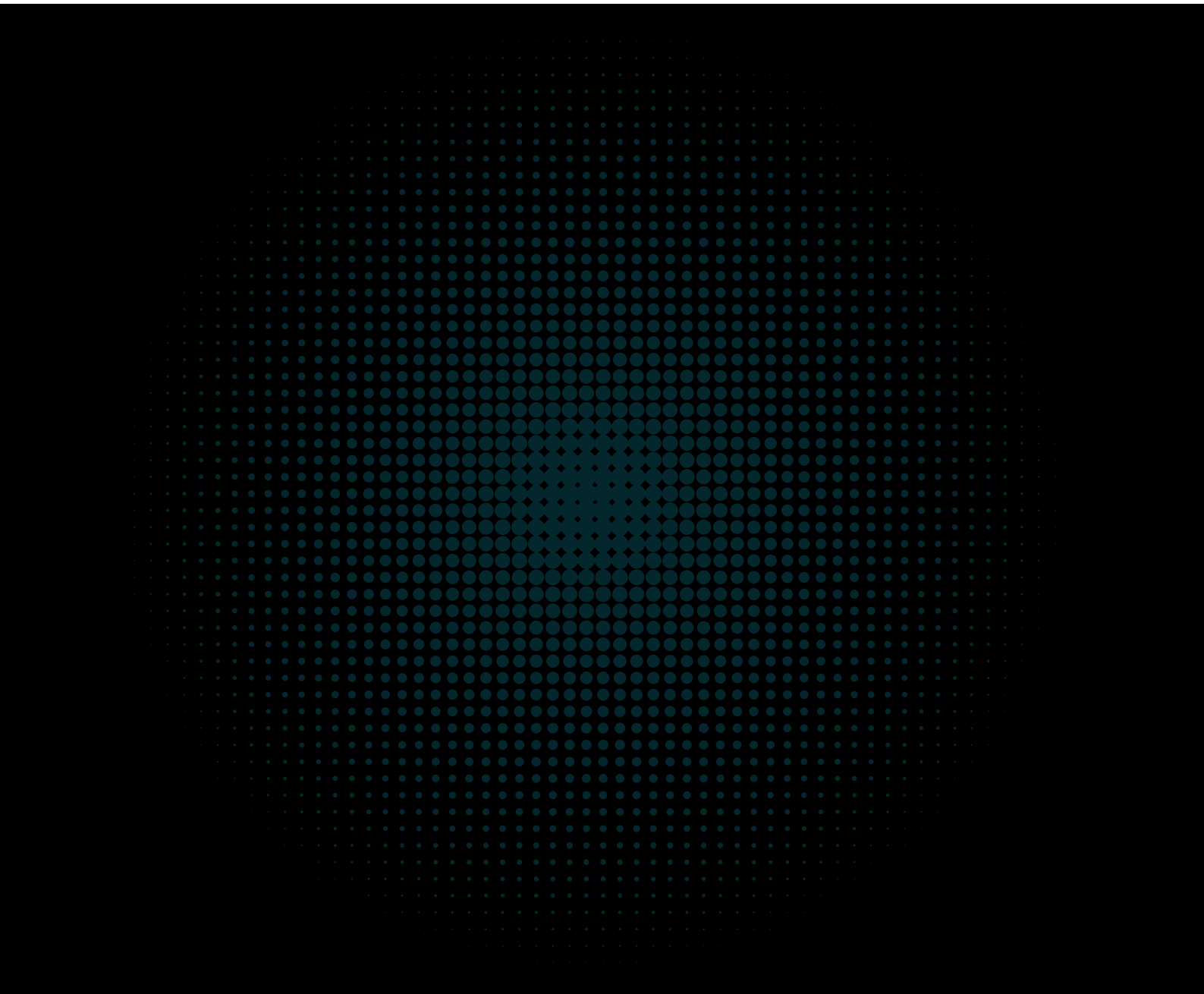
cmta.

CMTAT ACE Specification

CMTAT Functional Specifications to integrate Chainlink ACE (Automated Compliance Engine)

CMTAT Chainlink ACE: v0.3.0
First published: June 2026



CMTAT ACE integration project

Introduction

This project integrates a **CMTAT** security token with **Chainlink ACE** so that the token's compliance rules are defined in on-chain policies, which the policy engine evaluates on each protected operation. An issuer can adjust who can transact and under which conditions (KYC/allowlists, sanctions screening, transfer and volume limits, trading-hours windows, pause, reserve-backed minting) by reconfiguring those policies, rather than redeploying or modifying the token's business logic.

The problem it solves

Regulated tokens (security tokens, real-world assets (RWA), and stablecoins) need to enforce compliance rules such as eligibility, limits, freezes, and pauses, and those rules change over time as regulation, jurisdictions, or counterparties evolve. When the rules are embedded in the token, each change requires a contract upgrade or redeploy. This integration keeps the rules in a separate policy engine, so updating compliance is a configuration change rather than a code change.

The two building blocks

- **CMTAT (CMTA Token)** — an open security-token framework from the Capital Markets and Technology Association. It provides the ERC-20 token plus compliance modules: conditional transfers, account freeze / enforcement (ERC-7943), forced transfer & recovery, pause, in-contract documents, cross-chain mint/burn, and lifecycle controls.
- **Chainlink ACE (Automated Compliance Engine)** — a `PolicyEngine` that, for a protected function call, runs a configurable chain of **policies** (small contracts that approve or reject based on the call's parameters) and returns a decision. Policies are added, removed, and reordered by governance at runtime.

How it works

1. A token function (e.g. `transfer`, `mint`) is **protected**: before it takes effect, the token asks the PolicyEngine to evaluate the call.
2. An **extractor** decodes the call's calldata into named parameters (`from`, `to`, `amount`, ...).
3. The PolicyEngine runs the **policies** attached to that function's selector (pause, role-based access, sanctions/allowlist screening, volume/rate limits, reserve checks, and so on). If any policy rejects, the call reverts; otherwise it proceeds.
4. To change compliance behavior, governance attaches/detaches/reorders policies; **no token redeploy is needed**.

What you get

Two ready-to-deploy variants (standalone and upgradeable proxies), so an issuer can choose *how much* of compliance to externalize:

- **Lite** — keeps CMTAT's native role-based access control and uses ACE only for **transfer validation** (it replaces CMTAT's RuleEngine). Closest to a standard CMTAT token.

- **Standard — policy-authoritative**: ACE gates *all* state-changing operations (mint, burn, transfer, enforcement, admin) instead of local `onlyRole` checks; access control itself becomes a policy concern.

Compliance itself is expressed with **policies from the Chainlink ACE policy library** (for example pause, role-based access control, volume / rate / interval limits, and reserve-backed Proof-of-Reserve minting) that the issuer attaches to the token and configures. On top of those, this repo adds the glue needed to use them with CMTAT: a custom `TransferValidationPolicy` that reuses CMTAT's existing `IRule` transfer rules (KYC/sanctions/allowlist) as ACE policies, the **extractors** that map each token function's calldata to policy parameters, a complete deployment **demo**, and a deployment **preflight** check that catches common misconfigurations before going live.

Who it's for

Issuers of **security tokens, real-world assets (RWA), and stablecoins** (and their integrators) who want CMTAT's token feature set with compliance that can evolve through governance-controlled policy configuration rather than contract upgrades. (For example, a stablecoin can gate issuance with the reserve-backed `SecureMintPolicy` and screen holders with sanctions policies, while an RWA fund can enforce eligibility, transfer limits, and trading-hours windows.)

Example workflow: a transfer screened by a freeze policy

The sequence below shows a `transfer` being evaluated by the PolicyEngine against a **freeze policy**: the token forwards the call to the engine, an extractor decodes the parameters, and the policy rejects the transfer if the sender (or recipient) is frozen — otherwise the transfer proceeds and balances are updated.

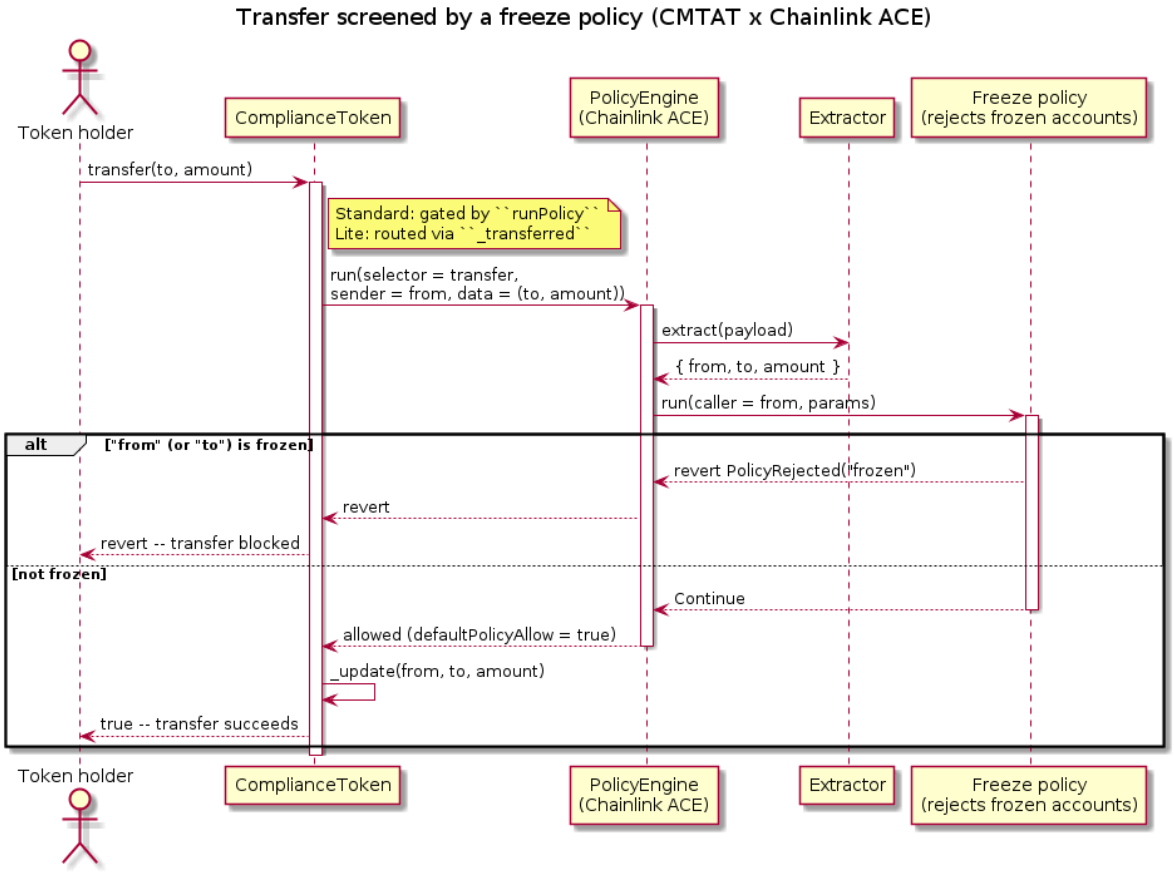


Diagram source: doc/img/transfer-freeze-workflow.puml.

> `defaultPolicyAllow = true` (shown in the diagram) is the engine's verdict for a call once its policies have run: ACE evaluates the attached policies and, if **none of them explicitly returned** `Allowed`, it falls back to this default. The policies shipped with this integration (pause, RBAC, transfer-validation, freeze, reserve-backed mint) **reject by reverting and otherwise return** `Continue` — they never return `Allowed` — so a call is permitted exactly when **no policy reverted** and the default is `true` (allow-by-default; reject only on an explicit policy revert). With `defaultPolicyAllow = false`, the same non-reverting chain would instead be **rejected** unless a terminal allow policy is attached. See Security Considerations.

Example workflow: reserve-backed minting (SecureMintPolicy + Chainlink Proof of Reserve)

`SecureMintPolicy` gates the `mint` selector against a **Chainlink Proof-of-Reserve (PoR)** feed: before issuing new tokens it reads the latest on-chain reserve value and rejects the mint if it would push `totalSupply` beyond the reserves (optionally adjusted by a configured margin), or if the feed is stale/negative. This enforces that the token can never be minted beyond what is backed by reserves — the reserve-backed issuance pattern used by stablecoins and RWAs.

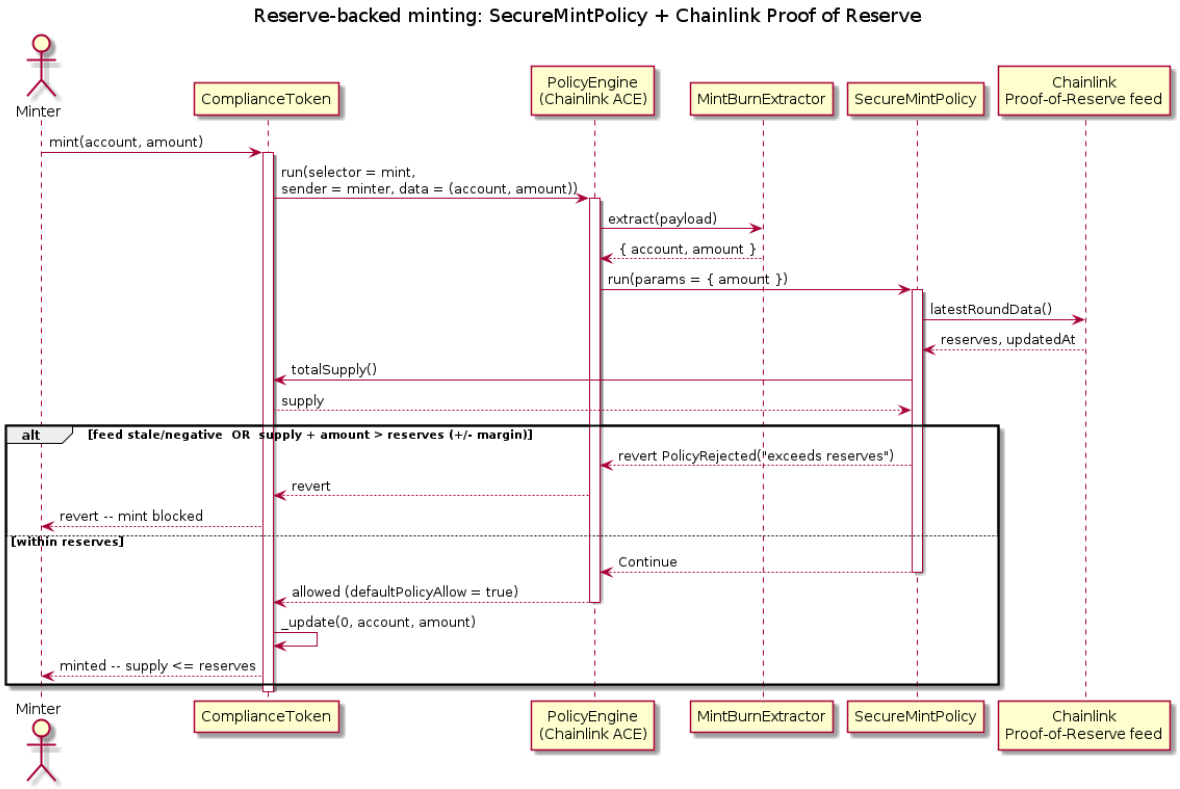


Diagram source: `doc/img/securemint-por-workflow.puml`.

Table of Contents

Deployment versions

Two versions are available:

- **Lite**: substitutes RuleEngine with Chainlink ACE PolicyEngine for transfer validation, while keeping CMTAT role-based module authorization.
- **Standard**: uses Chainlink ACE PolicyEngine as the authorization/compliance gate for state-changing operations, replacing local role-based authorization with policy checks.

Standard

Replaces CMTAT's `AccessControlUpgradeable` (role-based) with `OwnableUpgradeable` (single owner) and integrates Chainlink ACE `PolicyProtectedBaseUpgradeable` for access control and compliance validation on state-changing operations (mint, burn, transfer, enforcement, admin functions).

Contract	Proxy type
`ComplianceTokenCMTATStandalone`	None
`ComplianceTokenCMTATUpgradeable`	Transparent
`ComplianceTokenCMTATUUPSUpgradeable`	UUPS (`onlyOwner`)

Lite

Keeps CMTAT's `AccessControlUpgradeable` (role-based) for module authorization and adds Chainlink ACE PolicyEngine for transfer validation only, replacing CMTAT's RuleEngine.

Contract	Proxy type
`ComplianceTokenCMTATLiteStandalone`	None
`ComplianceTokenCMTATLiteUpgradeable`	Transparent
`ComplianceTokenCMTATLiteUUPSUpgradeable`	UUPS (`onlyRole(PROXY_UPGRADE_ROLE)`)

Changes from CMTAT

Warning (Standard Variant)

In the **Standard** variant, critical operations are authorized through ACE `runPolicy` checks instead of local `onlyRole(...)` checks. This includes core actions such as `mint`, burn functions, forced transfer/enforcement actions, and sensitive admin/configuration operations.

This means `PolicyEngine` configuration is security-critical infrastructure. A bad config change can unintentionally allow or block sensitive actions. It also introduces a direct runtime dependency on Chainlink ACE contracts (PolicyEngine, attached policies, extractor/mapper configuration): if ACE contracts are unavailable, misconfigured, or incorrectly upgraded, authorization and compliance checks in the token are directly affected. For `runPolicy` context handling, cleanup is best-effort on success only: context is cleared after the guarded function completes successfully. If the guarded call reverts, cleanup is not reached, and previously stored context remains in storage.

Treat the following as privileged governance actions:

- `addPolicy` / `removePolicy`
- `setExtractor` / `setPolicyMapper`
- `setDefaultAllow`
- `attachPolicyEngine`

Access Control

Aspect	CMTAT	Standard	Lite
Base model	`AccessControlUpgradeable` with 9+ roles	`OwnableUpgradeable` (single owner)	`AccessControlUpgradeable` (unchanged)
Authorization	`onlyRole(MINTER_ROLE)`, etc.	`runPolicy` modifier via PolicyEngine	`onlyRole()` for modules, PolicyEngine for transfers
Role management	`grantRole()` / `revokeRole()`	Managed externally via `RoleBasedAccessControlPolicy`	CMTAT roles preserved

Roles and permissions (Lite)

The **Lite** variant keeps CMTAT's native `AccessControlUpgradeable` roles. The table below lists each role and what it authorizes (verified against the module authorization hooks). All roles are managed by `DEFAULT_ADMIN_ROLE` via `grantRole` / `revokeRole`.

Role	Authorizes
`DEFAULT_ADMIN_ROLE`	Grant/revoke all roles; attach/detach the PolicyEngine (`attachPolicyEngine`, incl. `address(0)`); `deactivateContract`; `forcedTransfer`; ERC-20 attribute management (name/symbol/decimals); set the CCIP admin
`MINTER_ROLE`	`mint`
`BURNER_ROLE`	`burn`
`BURNER_FROM_ROLE`	Cross-chain burn from a holder (`crosschainBurn`)
`BURNER_SELF_ROLE`	Cross-chain self-burn

Role	Authorizes
`CROSS_CHAIN_ROLE`	Act as the token bridge for cross-chain mint/burn (`_checkTokenBridge`)
`PAUSER_ROLE`	`pause` / `unpause`
`ENFORCER_ROLE`	Freeze/unfreeze addresses (address-level enforcement)
`ERC20ENFORCER_ROLE`	Token-level freeze: `setFrozenTokens`, `freezePartialTokens`, `unfreezePartialTokens`
`DOCUMENT_ROLE`	Manage ERC-1643 documents (`setDocument` / `removeDocument`)
`EXTRA_INFORMATION_ROLE`	Set tokenId / terms / information
`PROXY_UPGRADE_ROLE`	Authorize UUPS upgrades (UUPS variant only)

The **Standard** variant does not use these roles for module access: it is `OwnableUpgradeable` (single `owner`) and routes authorization through the PolicyEngine, where role-like permissions are configured externally via `RoleBasedAccessControlPolicy` rather than `grantRole`.

Validation & Compliance

Aspect	CMTAT	Standard	Lite
Validation layer	`CMTATBaseRuleEngine` → `ValidationModuleRuleEngine`	`PolicyProtectedBaseUpgradeable` → `IPolicyEngine`	`ValidationModulePolicyEngine` → `IPolicyEngine`
Engine type	RuleEngine (custom interface)	Chainlink ACE PolicyEngine	Chainlink ACE PolicyEngine
Transfer check	`_canTransferGenericByModuleAndRevert()` + RuleEngine	PolicyEngine `run()` via `runPolicy` modifier	`_canTransferGenericByModuleAndRevert()` + PolicyEngine `run()`
ERC-1404 support	Via `ValidationModuleERC1404`	Not applicable (no module-level checks)	Via `PolicyValidationModuleERC1404`

In the **Lite** variant the ERC-1404 view is PolicyEngine-aware: after the module checks (pause/deactivate/freeze/active-balance) pass, `detectTransferRestriction` / `detectTransferRestrictionFrom` consult the PolicyEngine and return restriction code `7` (`TRANSFER_REJECTED_BY_POLICY_ENGINE_CODE`, message `"PolicyEngine:transferRejected"`) when the engine would reject the transfer. Module-level codes take precedence, and the view never reverts.

Detaching the PolicyEngine (Lite only)

The PolicyEngine can be detached on the **Lite** variant but not on the **Standard** variant. On Lite, an admin (`DEFAULT_ADMIN_ROLE`) may call `attachPolicyEngine(address(0))`: access control is CMTAT role-based and the engine is used for transfer validation only, so detaching simply disables ACE policy validation while CMTAT's native validation (pause, enforcement, allowlist, ...) stays in force. On **Standard**, detaching is rejected (`attachPolicyEngine(address(0))` reverts): authorization is policy-authoritative (every privileged operation is `runPolicy`-gated), so a zero engine would brick the token. This is enforced in `_validatePolicyEngine`, which keeps the non-zero requirement on Standard and relaxes it on Lite.

Initialization

The `Engine` struct parameter is replaced with a single `address policyEngine_`:

```
// CMTAT
constructor(forwarder, admin, ..., ICMTATConstructor.Engine memory engines_)

// ComplianceTokenCMTAT (Standard & Lite)
constructor(admin, ..., address policyEngine_)
```

Document management is handled in-contract via CMTAT's `DocumentERC1643Module` (no external document engine), and the snapshot engine has been removed from this integration, so neither a `documentEngine_` nor a `snapshotEngine_` parameter is taken.

ERC-2771 (gasless transaction forwarding) has been removed from all deployment contracts. The standalone contracts no longer take a `forwarderIrrevocable` parameter, and the upgradeable contracts have parameterless constructors.

Modules

All CMTAT functional modules are preserved in both variants:

- ERC20MintModule, ERC20BurnModule
- ERC20EnforcementModule (freeze/enforcement)
- PauseModule (Standard: `pause()` / `unpause()` / `deactivateContract()` are not exposed on the token; pausing is enforced externally via a PausePolicy on the PolicyEngine which rejects operations when paused; Lite: native `onlyRole(PAUSER_ROLE)`)
- DocumentERC1643Module (in-contract ERC-1643 document management, `DOCUMENT_ROLE`)
- ExtraInformationModule
- ERC20CrossChainModule, CCIPModule

> The external `DocumentEngineModule` and the `SnapshotEngineModule` from CMTAT are **not** used in this integration: documents are managed in-contract via `DocumentERC1643Module`, and snapshot support has been removed.

Removed from Standard

- `CMTATBaseAccessControl` — replaced by `OwnableUpgradeable`
- `AccessControlModule` — role management removed from contract
- `CMTATBaseRuleEngine` — replaced by `PolicyProtectedBaseUpgradeable`
- `ValidationModuleRuleEngine` — replaced by direct PolicyEngine calls
- All `onlyRole()` authorization functions — replaced by `runPolicy` modifier
- `pause()`, `unpause()`, `deactivateContract()` — not exposed on the token contract; the `_authorizePause` and `_authorizeDeactivate` hooks are intentionally left unimplemented so these functions remain abstract and are excluded from the compiled contract. Pausing is enforced externally via a PausePolicy attached to the PolicyEngine, which rejects protected operations when paused

Design notes

Why `approve()` is not policy-protected

`approve()` is intentionally not gated by `runPolicy` in either variant. An approval by itself does not move tokens; it only sets an allowance. The actual token movement happens via `transferFrom()`, which **is** policy-protected. Protecting `approve()` would add gas overhead without security benefit, since:

1. A malicious or excessive approval has no effect until `transferFrom()` is called, at which point the PolicyEngine validates the transfer.
2. The `ERC20TransferFromExtractor` extracts the `spender` address from `transferFrom()` calls, so policies can restrict which spenders are allowed to move tokens regardless of existing approvals.
3. In the Lite variant, `approve()` is gated by `whenNotPaused` as a convenience (matching upstream CMTAT behavior), but this is not a security-critical check.

SecureMintPolicy and cross-chain (Proof-of-Reserve) tokens

`SecureMintPolicy` enforces `mintAmount + totalSupply() <= reserves`, where `totalSupply()` is the **per-chain** supply of the token contract it is attached to. This is correct for a single-chain token, but is a footgun for a **cross-chain / bridgeable** token (`ERC20CrossChainModule` / `crosschainMint`):

- A `crosschainMint` on chain B mints tokens that were **burned on chain A**, so global supply does not increase, only chain B's local supply does.
- If the Proof-of-Reserve feed reports the **global** reserves backing the **global** supply, but the policy compares them against chain B's **local** `totalSupply()`, then as chain B's local supply approaches the global reserve value, **legitimate cross-chain mints will be rejected** (`"mint would exceed available reserves"`) even though the bridged tokens are fully backed.
- Conversely, applying a per-chain reserve value against each chain independently can **permit over-minting** of the global supply.

Guidance for issuers:

- The Proof-of-Reserve must validate the **whole multi-chain supply against the whole reserve**, not a single chain in isolation. Use a PoR feed that reports global reserves **and** a supply accounting that aggregates supply across all chains (e.g. a cross-chain aggregator / CCIP-based PoR), or do not gate `crosschainMint` with a per-chain `SecureMintPolicy`.
- The demo intentionally wires `SecureMintPolicy` to `mint()` only (genuine new issuance), **not** to `crosschainMint()`. Do not attach a naive per-chain `SecureMintPolicy` to `crosschainMint` unless your PoR design accounts for cross-chain supply as described above.

Removed from both variants

- `ERC2771Module` — gasless transaction forwarding is not supported (ACE does not currently support ERC-2771)

Added

- `PolicyProtectedBaseUpgradeable` — Chainlink ACE integration with ERC-7201 storage, `runPolicy` modifier, and policy engine lifecycle management
- `ValidationModulePolicyEngine` (Lite) — hybrid validation combining CMTAT module checks with PolicyEngine
- `PolicyValidationModuleERC1404` (Lite) — ERC-1404 transfer restriction codes with PolicyEngine awareness
- `TransferValidationPolicy` — Chainlink ACE policy that validates transfers using CMTAT's `IRule` interface (see TransferValidationPolicy below)
- `ERC20TransferFromExtractor` — Extractor that produces 4 parameters (`spender`, `from`, `to`, `amount`) for `transfer()` and `transferFrom()`
- `CrossChainMintBurnExtractor` — Extractor that maps `crosschainMint` / `crosschainBurn` into the `[from, to, amount]` layout so the same `IRule` rules can screen cross-chain issuance/redemption
- `CCTVersionModule` — overrides CMTAT's `VersionModule` so the token's `version()` returns the CMTAT-ACE integration release (currently `0.3.0`) instead of the underlying CMTAT framework version. The `version()` view follows the ERC-3643 and ERC-8303 (Contract Version) convention; see `doc/ERCSpecification/erc-8303.md`

Compliance Policies

Compliance behavior is expressed as **policies** attached to the PolicyEngine per `(token, function-selector)` pair. When a protected function runs, the engine evaluates the policies registered for that selector, feeding each one the parameters produced by the **extractor** configured for that selector. A policy either lets evaluation continue, short-circuits to "allow", or reverts to reject the call.

This repository ships one custom policy (`TransferValidationPolicy`) and reuses the policy library from `@chainlink/ace`. The most useful policies for a token issuer are summarized below; each row links to the integration test that demonstrates it against a ComplianceToken.

Policies used and tested in this repo

Policy	What it enforces	`run` parameters (via extractor)	Example use case
`TransferValidationPolicy` (custom)	Runs an array of CMTAT `IRule` contracts; rejects on any non-zero restriction code	`[from,to,amount]` or `[spender,from,to,amount]`	Reuse existing CMTAT transfer-restriction rules (sanctions/KYC/max-amount) as ACE policies; see TransferValidationPolicy. Tests: `test/custom/transferValidationPolicy.test.js`, `crosschainScreening.test.js`, `mintBurnScreening.test.js`
`PausePolicy`	Rejects protected calls while paused	none	Emergency pause of mint/burn/transfer without a pause role on the token (Standard variant). Tests: `test/common/ace/PausePolicyCommon.js`
`RoleBasedAccessControlPolicy`	Caller must hold the role mapped to the selector	none (uses caller + selector)	Externalized role management for admin/lifecycle operations (Standard variant). Tests: `test/common/ace/RBACPolicyCommon.js`
`SecureMintPolicy`	`mintAmount + totalSupply() <= reserves` from a Chainlink PoR feed	`[amount]`	Reserve-backed (Proof-of-Reserve) minting. Tests: `test/custom/secureMintPolicy.test.js`. See the cross-chain PoR caveat
`MaxPolicy`	Per-call hard cap (non-accumulating)	`[amount]`	Maximum amount per single transfer/mint. Tests: `test/custom/maxPolicy.test.js`
`VolumePolicy`	Per-call `min <= amount <= max`	`[amount]`	Minimum and maximum ticket size per operation. Tests: `test/custom/volumePolicy.test.js`

Policy	What it enforces	`run` parameters (via extractor)	Example use case
`VolumeRatePolicy`	Per-account cumulative cap within a rolling time window	`[amount, account]`	Rate-limit how much each holder can move per day/hour. Tests: `test/custom/volumeRatePolicy.test.js`
`IntervalPolicy`	Execution allowed only within a time window of a repeating cycle	none	Trading-hours / settlement-window restriction. Tests: `test/custom/intervalPolicy.test.js`
`OnlyOwnerPolicy`	Caller must be the policy's owner	none (uses caller)	Funnel a sensitive function (e.g. `mint`) through a single governance key, layered on top of CMTAT roles. Tests: `test/custom/onlyOwnerPolicy.test.js`

Other policies available from `@chainlink/ace`

These are part of the installed `@chainlink/ace` policy library and can be wired the same way, but are not currently configured or tested in this repository:

Policy	What it enforces
`BypassPolicy`	If an extracted address is on a bypass list, returns `Allowed` (short-circuits evaluation to allow). The only bundled policy that returns a terminal allow; required if you operate with `defaultAllow = false` (see below)
`AllowPolicy`	Allow-list: rejects unless the extracted address is on the list
`RejectPolicy`	Block-list: rejects if the extracted address is on the list
`OnlyAuthorizedSenderPolicy`	Rejects unless the caller is on an authorized-sender list
`OnlySubjectOwnerPolicy`	Caller must be the `Ownable` owner of the token (subject); fits the Standard variant's `OwnableUpgradeable`
`CertifiedActionValidatorPolicy` / `…DONValidatorPolicy` / `ERC20TransferValidatorPolicy`	Validate off-chain "certified actions" (e.g. DON-signed approvals) before allowing an operation; advanced, for attestation-gated flows

How policies combine (important)

- **Per-selector:** a policy attached to `transfer` does **not** apply to `mint`, `forcedTransfer`, or `crosschainMint`. Attach screening to every movement selector you care about. See Policy-Protected Functions and the `MintBurnExtractor` / `CrossChainMintBurnExtractor` extractors.
- **Allow-by-default model:** nearly all of the policies above return `Continue` on success and only **revert** to reject; none of them return a terminal `Allowed` except `BypassPolicy`. With the PolicyEngine's `defaultAllow = true`, a call is allowed unless some policy reverts. With `defaultAllow = false`, a call is rejected **unless** some policy returns `Allowed`, so a fail-closed deployment requires `BypassPolicy` (or a custom terminal-allow policy) on every

protected selector, otherwise operations are bricked. Use the policy preflight check to verify this before going live.
- **Ordering:** policies execute in attachment order; the first `Allowed` short-circuits, any revert rejects. Put fail-closed/restrictive checks before any bypass.

TransferValidationPolicy

`TransferValidationPolicy` is a Chainlink ACE policy that bridges CMTAT's `IRule` interface with the PolicyEngine, enabling reuse of existing transfer restriction rules as ACE policies.

How it works

The policy accepts an array of `IRule` contracts. When the PolicyEngine invokes the policy during a `transfer()` or `transferFrom()`, each rule is evaluated in order. If any rule returns a non-zero restriction code, the policy reverts with `PolicyRejected` containing the rule's human-readable message.

It supports two extractor layouts:

Extractor	Parameters	Used by
`ERC20TransferExtractor`	`[from, to, amount]`	Calls `detectTransferRestriction(from, to, amount)`
`ERC20TransferFromExtractor`	`[spender, from, to, amount]`	Calls `detectTransferRestrictionFrom(spender, from, to, amount)`

`run` vs `postRun`: view validation and stateful enforcement

CMTAT's native flow calls only `IRule.transferred(...)` on each rule during a transfer (enforcement **and** any state update happen there); `detectTransferRestriction*` is the read-only preview. ACE cannot fuse the two, because the engine reuses the policy's `run()` for **both** the read-only preview (`check()`, which is STATICCALLed by `canTransfer` / ERC-1404 `detectTransferRestriction` / off-chain simulations) and the state-flow pre-check. A STATICCALL forbids state writes, so `run()` must be `view`. `TransferValidationPolicy` therefore splits the work:

Hook	When the PolicyEngine calls it	What it does
`run` (`view`)	on every `check()` **and** every `run()` (state)	validates with the view `detectTransferRestriction*`; reverts `PolicyRejected` if any rule returns a non-zero code
`postRun` (state)	**only** after a successful `run` on the **state** path (`run(payload)`); **never** on `check()`	calls each rule's state-mutating **`transferred(...)`** hook — this advances **stateful** rules (rolling-window volume caps, per-period counters, conditional rules) and applies their on-chain enforcement

Consequences:

- **The view check in `run` is required, not redundant.** Without it, `canTransfer` / `detectTransferRestriction` would only run a no-op preview and report a transfer as allowed even when it would revert — breaking the ERC-7943 / ERC-1404 view contract and any wallet/AMM/custodian that simulates via `check()`.
- **Stateful rules are enforced via `postRun` / `transferred`**, exactly mirroring CMTAT's write path (`RuleEngine.transferred` → `rule.transferred` per rule). `postRun` runs once per **executed** transfer, never on a preview, so a `canTransfer` simulation has no side effects.
- For correct behavior a rule's `detectTransferRestriction*` and `transferred` should be **consistent** (the CMTA Rules library rules are): `run` then provides an accurate preview/early veto and `postRun` the authoritative state-path enforcement. The integration applies **both**, so it is never weaker than CMTAT and works whether a rule's logic lives in `detect*`, in `transferred`, or in both.

Mock rules

Mock `IRule` implementations are provided in `contracts/modules/chainlink-ace/mocks/TransferRuleMocks.sol` for testing and demonstration:

- **`MaxAmountRule`** — Rejects transfers where the amount exceeds a configurable maximum (restriction code `13`). Stateless.
- **`RestrictedAddressRule`** — Rejects transfers involving addresses on a configurable restricted list (codes `14` / `15` for sender/recipient). Stateless.
- **`CumulativeCapRule`** — **Stateful**: caps the cumulative amount each sender may send; the `sent[from]` counter is advanced by `transferred` (state path) and read by `detect*`. Demonstrates that stateful enforcement requires `postRun` / `transferred` (the NM-2 path).
- **`TransferredEnforcedCapRule`** — **Stateful**: `detect*` is permissive (always OK) and enforcement lives **solely** in `transferred`, exactly as CMTAT's RuleEngine does — proving `postRun` / `transferred` is an authoritative enforcer, not just a state-updater.

Setup

1. Deploy the extractor and set it on the PolicyEngine:

```
const extractor = await ethers.deployContract('ERC20TransferFromExtractor');
const transferSelector =
cmtat.interface.getFunction('transfer(address,uint256)').selector;
const transferFromSelector = cmtat.interface.getFunction(
  'transferFrom(address,address,uint256)',
).selector;

await policyEngine.setExtractor(transferSelector, await
extractor.getAddress());
await policyEngine.setExtractor(transferFromSelector, await
extractor.getAddress());
```

2. Deploy rule contracts and the policy:

```
const maxAmountRule = await ethers.deployContract('MaxAmountRule', [1000n]);
const restrictedRule = await ethers.deployContract('RestrictedAddressRule',
[[]]);
```

```
const configParams = abiCoder.encode(
  ['address[]'],
  [[await maxAmountRule.getAddress(), await restrictedRule.getAddress()]],
);

const policy = await upgrades.deployProxy(
  await ethers.getContractFactory('TransferValidationPolicy'),
  [policyEngineAddress, adminAddress, configParams],
  {
    initializer: 'initialize',
    unsafeAllow: ['constructor', 'missing-initializer', 'missing-initializer-
call'],
  },
);
```

3. Register the policy for transfer selectors with parameter names:

```
const PARAM_SPENDER = keccak256(toUtf8Bytes('spender'));
const PARAM_FROM = keccak256(toUtf8Bytes('from'));
const PARAM_TO = keccak256(toUtf8Bytes('to'));
const PARAM_AMOUNT = keccak256(toUtf8Bytes('amount'));

await policyEngine.addPolicy(cmtatAddress, transferSelector, policyAddress, [
  PARAM_SPENDER,
  PARAM_FROM,
  PARAM_TO,
  PARAM_AMOUNT,
]);
await policyEngine.addPolicy(cmtatAddress, transferFromSelector, policyAddress,
[
  PARAM_SPENDER,
  PARAM_FROM,
  PARAM_TO,
  PARAM_AMOUNT,
]);
```

4. Rules can be updated at any time by the policy owner:

```
await policy.setRules([newRuleAddress1, newRuleAddress2]);
```

Writing custom rules

Implement the `IRule` interface to create custom transfer restriction logic:

```
contract MyCustomRule is IRule {
  function detectTransferRestriction(
    address from,
    address to,
    uint256 amount
  ) public view override returns (uint8) {
    // Return 0 for allowed, non-zero for rejected
  }

  function detectTransferRestrictionFrom(
```

```
    address spender,
    address from,
    address to,
    uint256 amount
  ) public view override returns (uint8) {
    // Validate spender + transfer params
  }

  function messageForTransferRestriction(
    uint8 code
  ) external pure override returns (string memory) {
    // Return human-readable rejection reason
  }

  // ... canTransfer(), canReturnTransferRestrictionCode()
}
```

Using the official CMTA Rules library

You do not have to write rules from scratch: the ready-made rules from CMTA's Rules library (allowlist/whitelist, blacklist, sanctions, conditional transfer, etc.) implement the same `IRule` interface and can be reused with this integration in two ways:

- **Through `TransferValidationPolicy`** — pass the deployed CMTA rule addresses in the policy's rule array (at `initialize` or via `setRules`). The policy runs each `IRule` and rejects on any non-zero restriction code, so no rule code changes are needed. This is the recommended path and is what the mock rules demonstrate.
- **By extending `RuleEngine` with the `IPolicy` interface** — wrap CMTA's RuleEngine (which already aggregates and orchestrates a set of `IRule` contracts) in a Chainlink ACE `Policy` so the whole RuleEngine becomes a single ACE policy. Use this when you want to keep the RuleEngine's rule-management and orchestration logic rather than re-listing individual rules on `TransferValidationPolicy`.

Both approaches let an issuer reuse the audited CMTA rule set while still gating transfers through the Chainlink ACE PolicyEngine.

ERC-165 Interface Support

This integration includes ERC-165 interface discovery for both the protected token side and policy side:

- **Protected-token interface support**: `PolicyProtectedBaseUpgradeable` exposes `IPolicyProtected` via `supportsInterface`, and the Standard/Lite token bases propagate that support through their own `supportsInterface` overrides.
- **Policy interface support**: `TransferValidationPolicy` extends Chainlink ACE `Policy`, and `Policy` exposes `IPolicy` via ERC-165.
- **Rule interface support in mocks**: the included `TransferRuleMocks` expose `IRule` via `supportsInterface` for compatibility testing.

This allows integrators and tooling to programmatically verify interface compatibility before wiring policies, engines, and rule contracts together.

ERC-7943 (uRWA) support

Both variants advertise the ERC-7943 (uRWA) **fungible** interface id `0x3edbb4c4` via `supportsInterface`, and implement its check/enforcement surface:

- `forcedTransfer`, `setFrozenTokens`, `getFrozenTokens` — enforcement (from CMTAT's `ERC20EnforcementModule`).
- `canTransfer(from,to,amount)`, `canSend(account)`, `canReceive(account)` — non-reverting view checks. `canTransfer` combines the unfrozen-balance check, `canSend`/`canReceive`, and the PolicyEngine's permissioned rules (queried via the read-only `check`, mapping a revert to `false`).

Notes:

- **Lite** uses CMTAT's account freeze, so `canSend`/`canReceive` return `false` for a frozen account.
- **Standard** has no on-chain account allowlist/freeze on the token (send/receive eligibility is decided per transfer by the PolicyEngine inside `canTransfer`), so `canSend`/`canReceive` report no token-level account restriction. The authoritative gate is `canTransfer`.

Conformance is covered by `test/custom/erc7943Compliance.test.js`.

Library

- CMTAT v3.3.0-rc1
- Chainlink ACE `1.1.1`
- OpenZeppelin Contracts `5.6.1`
- OpenZeppelin Contracts Upgradeable `5.6.1`

Initialize submodules

```
git submodule update --init --recursive
```

Install dependencies

You can use any package manager either npm, yarn or pnpm. For example you can type:

```
bun install
```

Compile contracts

To compile

```
bunx hardhat compile
```

Testing

To run tests:

```
bunx hardhat test
```

Linting & Formatting

ESLint

Lint JavaScript files (tests, scripts, config):

```
bun run lint
```

Auto-fix fixable issues:

```
bun run lint:fix
```

Prettier

Check formatting for JS, JSON, Markdown, and Solidity:

```
bun run format:check
```

Auto-format all files:

```
bun run format
```

Solidity formatting uses prettier-plugin-solidity and is scoped to `contracts/**/*.sol` only (submodules and dependencies are excluded).

Scripts

Deployment scripts

> 📖 **Read the Deployment Guide first.** It explains how the scripts work, the risks (selector-coverage completeness, `defaultPolicyAllow`, atomic proxy init, extractor/parameter matching, Standard vs Lite responsibilities, roles), and a checklist for using these scripts — or writing your own — without bricking the token or leaving a privileged selector ungated. Always finish with the preflight check.

Individual deployment scripts are available for each contract variant:

Script	Description
`scripts/lite/deploy-lite-standalone.js`	Lite standalone (no proxy)
`scripts/lite/deploy-lite-upgradeable.js`	Lite transparent proxy
`scripts/lite/deploy-lite-uups.js`	Lite UUPS proxy
`scripts/standard/deploy-standard-standalone.js`	Standard standalone (no proxy)
`scripts/standard/deploy-standard-upgradeable.js`	Standard transparent proxy
`scripts/standard/deploy-standard-uups.js`	Standard UUPS proxy

Run any script with:

```
bunx hardhat run scripts/lite/deploy-lite-standalone.js
```

Demo script

`scripts/demo.js` provides a complete end-to-end deployment of the Standard variant with the full Chainlink ACE policy stack. It deploys and wires together all contracts in the correct order:

1. **PolicyEngine** (proxy) — central policy orchestrator with `defaultAllow = true`
2. **ComplianceTokenCMTATStandalone** — the token contract, attached to the PolicyEngine
3. **PausePolicy** (proxy) — added to all state-changing selectors (mint, burn, transfer, enforcement, admin)
4. **RoleBasedAccessControlPolicy** (proxy) — added to admin selectors with role-to-selector mappings
5. **MockV3Aggregator** — mock Chainlink reserve price feed (Hardhat network only)
6. **SecureMintPolicy** (proxy) — added to `mint()`, enforces reserve-backed minting via price feed
7. **MintBurnExtractor** — set for `mint()` selector, extracts `account` and `amount` parameters
8. **ERC20TransferExtractor** — set for `transfer()` selector
9. **ERC20TransferFromExtractor** — set for `transferFrom()` selector
10. **MaxAmountRule** + **RestrictedAddressRule** — mock IRule contracts for transfer validation
11. **TransferValidationPolicy** (proxy) — added to `transfer()` and `transferFrom()` with both rules

Documents are managed in-contract via `setDocument()` (`DocumentERC1643Module`, `DOCUMENT_ROLE`); there is no external document or snapshot engine.

The script also configures RBAC operation allowances and grants roles (`MINTER_ROLE`, `BURNER_ROLE`, `BURNER_FROM_ROLE`, `ENFORCER_ROLE`, `ERC20ENFORCER_ROLE`, `DOCUMENT_ROLE`) to the admin account.

Policy execution order per function:

- `mint()` → PausePolicy → RBAC → SecureMintPolicy
- `transfer()` / `transferFrom()` → PausePolicy → TransferValidationPolicy
- All other state-changing functions → PausePolicy → RBAC

Run the demo on a local Hardhat network:

```
bunx hardhat run scripts/demo.js
```

Policy preflight check

`scripts/preflight.js` verifies that a deployed token will **not** have its state-changing operations bricked by the PolicyEngine configuration, and prints per-selector policy coverage.

> **Important:** This integration is designed for **`defaultAllow = true`**. The bundled policies (`PausePolicy`, `RoleBasedAccessControlPolicy`, `TransferValidationPolicy`) return `Continue`, never `Allowed`, so the engine always falls through to the default. With `defaultAllow = false`, **every** policy-routed operation (mint/burn/transfer/...) reverts (even selectors that have policies attached), and the token is effectively frozen. The token must also be **attached** to the engine. The preflight reconstructs the effective `defaultAllow`

> (global + per-target) and attachment state from on-chain events and **exits non-zero** if the token would be bricked, so it can gate a deployment pipeline.

```
POLICY_ENGINE=0x... TOKEN=0x... \
  [TOKEN_CONTRACT=ComplianceTokenCMTATLiteStandalone] \
  bunx hardhat run scripts/preflight.js --network <network>
```

`TOKEN_CONTRACT` defaults to `ComplianceTokenCMTATStandalone`; set it to the Lite artifact when checking a Lite deployment. The invariant tests in `test/deployment/preflightPolicyCoverage.test.js` assert the preflight verdict matches real on-chain behavior.

Audit Reports Summary

This section summarizes the static-analysis reports available in this repository.

> ⚠️ **This project has NOT been formally audited by a security company. Use at your own risk.** The reports below are automated static-analysis (Slither, Aderyn) and AI-generated reviews only — they are **not** a substitute for a professional manual security audit. Do not deploy to production with real value without an independent, professional audit.

> 🔒 **See** `doc/audits/AUDIT_OVERVIEW.md` for the security overview: the latest Slither and Aderyn results (v0.3.0, mocks included), the AI/internal audit findings that were fixed, and the per-tool dispositions. **Latest static analysis: 0 High to fix** — every tool finding is a false positive, an intentional design choice, environment/mock noise, or cosmetic.

Slither

Here is the list of report performed with Slither

```
slither . --checklist > doc/audits/tools/v0.2.0/slither-report.md
```

`bun run slither` generates timestamped reports in the `reports/` directory:

- **JSON** — `reports/slither-report-<timestamp>.json`
- **Markdown** — `reports/slither-report-<timestamp>.md`

The direct `slither ... --checklist` command above writes a checklist-style report to `doc/audits/tools/v0.2.0/slither-report.md`.

Version	Report	Assessment
v0.3.0	slither-report.md	slither-report-feedback.md
v0.2.0	slither-report.md	slither-report-feedback.md
v0.1.0	slither-report.md	slither-report-feedback.md

Latest (v0.3.0, **mocks included**): **0 High · 11 Medium · 10 Low · 21 Informational** — nothing to fix. The per-finding breakdown and dispositions are in `doc/audits/AUDIT_OVERVIEW.md` and the feedback file.

Aderyn

Here is the list of report performed with Aderyn

```
# v0.3.0 — mocks INCLUDED (no -x mocks)
aderyn --output doc/audits/tools/v0.3.0/aderyn/aderyn-report.md
# earlier runs excluded mocks:
aderyn -x mocks --output doc/audits/tools/aderyn-report.md
```

Version	Report	Assessment
v0.3.0	aderyn-report.md	aderyn-report-feedback.md
current	aderyn-report.md	aderyn-report-feedback.md

Latest (v0.3.0, **mocks included**): **2 High · 11 Low** — nothing to fix (the Highs are the accepted-context / false-positive items). The per-finding breakdown and dispositions are in `doc/audits/AUDIT_OVERVIEW.md` and the feedback file.

Nethermind AuditAgent (AI review)

> ⚠️ **This report was generated entirely by AI and has not been manually reviewed by Nethermind's security team.** It does not constitute a security audit; findings may contain errors or omissions and must be independently verified.

AI-generated review of the v0.2.0 codebase (`doc/audits/tools/v0.2.0/nethermind-audit-agent/`). Developer triage: `audit_agent_report-feedback.md`. Outcome: **6 fixed in code, 2 accepted as documented design, 1 informational, 1 false positive** — all addressed in the v0.3.0 release.

ID	Finding	Tool sev	Our verdict	Status
NM-1	Uninitialized proxy hijack via public `initialize()`	High	Informational (not exploitable as deployed)	No code change (docs)
NM-2	`TransferValidationPolicy` never calls stateful `IRule.transferred()`	High	Accepted (High)	**Fixed**
NM-3	Unmapped inherited privileged selectors bypass policy auth (Standard)	Medium	Accepted as design (High; Lite already safe)	No code change; deployment remediated
NM-4	`MintBurnExtractor` lacks `burn(address,uint256)`	Low	Accepted (Medium)	**Fixed**
NM-5	`detectTransferRestriction*` reverts when frozen > balance	Low	Accepted	**Fixed**

ID	Finding	Tool sev	Our verdict	Status
NM-6	Context cleared mid-batch breaks batch ops	Low	Accepted	**Fixed**
NM-7	`burnAndMint` bypasses screening in Lite	Low	Accepted (Medium)	**Fixed**
NM-8	Standard `canSend` / `canReceive` always `true`	Info	Fixed (account-level signal added)	**Fixed**
NM-9	`initialize()` accepts zero PolicyEngine (Standard)	Info	**False positive**	Rejected
NM-10	Lite flows only screened if exact selectors wired	Info	Accepted by design	No code change; deployment + preflight

Coverage

Writes coverage files to *doc/coverage* using **solidity-coverage** hardhat plugin with config at **.solcover.js**

```
bunx hardhat coverage
npx hardhat coverage
```

Policy-Protected Functions (Current Integration)

This project now documents the policy-protected function selectors explicitly. The list below reflects the selectors wired in deployment/test flows (`scripts/demo.js`, `test/deploymentUtils.js`).

Core transfer selectors (Standard + Lite)

Function signature	Selector
`transfer(address,uint256)`	`0xa9059cbb`
`transferFrom(address,address,uint256)`	`0x23b872dd`

Admin/lifecycle selectors (Standard policy-authoritative flow)

Function signature	Selector
`mint(address,uint256)`	`0x40c10f19`
`burn(address,uint256)`	`0x9dc29fac`

Function signature	Selector
`burn(uint256)`	`0x42966c68`
`burnFrom(address,uint256)`	`0x79cc6790`
`forcedTransfer(address,address,uint256)`	`0x9fc1d0e7`
`freezePartialTokens(address,uint256)`	`0x125c4a33`
`unfreezePartialTokens(address,uint256)`	`0x1fe56f7d`
`setName(string)`	`0xc47f0027`
`setSymbol(string)`	`0xb84c8246`
`setTokenId(string)`	`0xdcfd616f`
`setDocument(bytes32,string,bytes32)`	`0x010648ca`
`setCCIPAdmin(address)`	`0xa8fa343c`
`crosschainMint(address,uint256)`	`0x18bf5077`
`crosschainBurn(address,uint256)`	`0x2b8c49e3`

Note: exact policy chains per selector (PausePolicy, RBAC, TransferValidationPolicy, etc.) can vary by deployment configuration.

Security Considerations

Standard variant is policy-authoritative: selector coverage is part of the deployment's security

In the **Standard** variant, the token delegates **all** authorization to the ACE PolicyEngine: every privileged operation is gated by `runPolicy`, which evaluates the policies wired for the function's **selector** (`msg.sig`). There is intentionally **no on-token role check** — the engine is the single, authoritative gate. (The **Lite** variant is different: it keeps CMTAT's native `onlyRole(...)` access control and uses the engine only for transfer validation, so its authorization does not depend on per-selector wiring.)

A direct consequence for the Standard variant is that **its safety is a property of the deployment configuration, not of the contract alone**. Two settings interact:

- **Selector coverage.** Authorization only applies to selectors that have a policy wired. The same privileged logic is reachable through several entrypoints with **different** selectors — overloads (`mint(address,uint256)` vs `mint(address,uint256,bytes)`), batch variants (`batchMint`, `batchBurn`), and multiplexers such as `burnAndMint(...)` (whose inner `burn`/`mint` run under the **`burnAndMint`** selector). Every such privileged selector must be wired, not only the canonical ones.
- **`defaultPolicyAllow`** — the engine's behavior for a selector that has **no** policy wired:
  - `defaultPolicyAllow = true` (allow-by-default): an **unwired** selector is **allowed**. This is convenient, but it means any privileged selector that was not wired (e.g. an

overlooked overload or `burnAndMint`) is callable without authorization. Under this setting you **must** wire a policy for *every* privileged selector.
    - `defaultPolicyAllow = false` (fail-closed): an **unwired** selector **reverts**. Forgetting to wire a selector becomes a safe failure (DoS) instead of an open door. Note the trade-off: the policies shipped here (`PausePolicy`, `RoleBasedAccessControlPolicy`, `TransferValidationPolicy`) return `Continue`, never `Allowed` — so under fail-closed you must also attach a **terminal allow** policy (or a policy that returns `Allowed`) on each selector you intend to permit, otherwise even correctly-wired operations revert.

Recommendation. For the Standard variant, choose **one** of:

1. Wire an access-control policy for **every** privileged selector — derived from the full token ABI, including the overloads/batch/multiplexer selectors above — and keep `defaultPolicyAllow = true`; **or**
2. Deploy with `defaultPolicyAllow = false` (fail-closed) plus an explicit terminal allow policy on each permitted selector.

Use the policy preflight check before going live to confirm coverage, and treat any unwired privileged selector as a deployment blocker.

FAQ for Issuers Using CMTAT with ACE Policies

> Warning: This FAQ is best-effort guidance for this repository integration. It may be incomplete and is not a substitute for official ACE documentation, legal advice, or a professional security review.

1. What does ACE add to CMTAT?

ACE moves compliance checks into separate policy contracts. This lets you update compliance rules without redeploying the token.

2. Do I still need CMTAT roles if ACE controls authorization?

Yes.

- Keep CMTAT roles where possible as a second safety layer, so a policy misconfiguration alone is less likely to enable sensitive actions.
- Treat ACE policy configuration as high-privilege admin control: changing policies, ordering, extractors, or `defaultAllow` can effectively allow or block critical token operations.

3. Which CMTAT version should I choose: lite or standard?

Use `lite` if you mainly need policy checks on transfers. Use `standard` if you also want policy checks on admin and lifecycle actions.

4. Who should own and manage the PolicyEngine?

Use a highly trusted governance setup, such as a multisig, DAO, or timelock. Whoever controls PolicyEngine settings effectively controls token compliance behavior.

5. What is the minimum policy set for a production issuer?

For token issuers, a common baseline is:

- Pause policy.
- Role-based access policy.
- Transfer restriction policy (for example KYC/sanctions/rule checks).
- A clearly defined default result (`defaultAllow=true` or `defaultAllow=false`).

6. Should default policy outcome be allow or reject?

Choose based on your operating model:

- `defaultAllow=true`: allow by default, and block only when a policy rejects.
- `defaultAllow=false`: reject by default, and allow only when policies explicitly allow.

In ACE, `true` is the usual default behavior; confirm and document your choice before launch.

7. How do I avoid policy ordering mistakes?

Start with restrictive checks, then business-limit checks, and place permissive/bypass behavior only where intentionally needed. A policy that returns `Allow` stops evaluation of later policies.

8. What happens if extractor or parameter mapping is wrong?

Policies may read the wrong values or fail unexpectedly. Treat extractor and parameter mapping as security-critical configuration, and test them like contract code.

9. Can I enforce different policies for transfer and transferFrom?

Yes. `transfer` and `transferFrom` use different selectors, so configure and test both paths separately. Include spender-specific checks for `transferFrom`.

10. How should I use context safely?

Use one of the two ACE patterns:

- Preferred for custom functions: pass `context` directly with `runPolicyWithContext(context)`.
- For fixed interfaces (like ERC-20 functions): call `setContext(...)` and consume it in the same atomic transaction.

Do not leave context pending across transactions.

11. What governance process should I use for policy changes?

Use a staged process:

1. Propose the change and simulate it in staging.
2. Review policy order, extractor mapping, and default outcome.
3. Execute through timelock/multisig.

4. Monitor events and transfer behavior after deployment.

12. What should I monitor in production?

Monitor:

- Policy add/remove actions.
- Extractor and mapping changes.
- `defaultAllow` changes (this flips the fallback behavior when all policies return `Continue`: `true` = allow, `false` = reject).
- Policy execution failures.
- Sudden increases in rejected or bypassed actions.

13. How do I prepare for regulator or auditor questions?

Maintain an audit-ready change log with policy versions, activation times, approval records, and test evidence for each policy update.

14. What are common integration mistakes?

- Wrong policy order (accidental early bypass).
- Missing extractor for a protected selector.
- Incorrect parameter names or mapping.
- No tests for revert/context behavior.
- Weak governance around PolicyEngine admin changes.

15. What should my pre-mainnet checklist include?

- Role/admin key setup completed.
- Policy chain and order reviewed.
- Extractor and parameter mapping tested for each selector.
- Default outcome verified for each contract.
- Pause and incident runbook tested.
- Upgrade and rollback plan approved.

16. How do I handle an incident (bad policy push or false rejects)?

Use an incident runbook with clear authority to pause sensitive actions, revert bad policy settings, communicate with counterparties, and re-enable flows in controlled phases.

17. Do I need separate testing for upgrades?

Yes. Run compliance regression tests for every upgrade, including policy-chain behavior, extractor decoding, and role/authorization invariants.

18. What documentation should I publish to integrators?

Publish a short integration guide that includes:

- Which functions are policy-protected (function names/selectors).
- What each policy does in normal operation.
- Common failure cases and the revert reasons integrators may see.

- How admin/policy changes are approved and announced.
- Who to contact for support and incident escalation.

License

This repository is licensed under the **Mozilla Public License 2.0 (MPL-2.0)** — see `LICENSE` — except for a few files that carry a different per-file `SPDX-License-Identifier`.

> **Note — mixed licensing, review before production use.** The following files are licensed under **BUSL-1.1 (Business Source License 1.1)**, inherited from the Chainlink ACE code they derive from, rather than MPL-2.0:
>
> - `contracts/modules/chainlink-ace/custom/MintBurnExtractor.sol`
> - `contracts/modules/chainlink-ace/custom/ERC20TransferFromExtractor.sol`
> - `contracts/modules/chainlink-ace/mocks/PolicyProtectedUpgradeableMocks.sol`
>
> The **Chainlink ACE** dependency (`@chainlink/ace` — the `PolicyEngine` and the bundled policies) is also BUSL-1.1. BUSL-1.1 is a *source-available* license, not an OSI open-source license: it can restrict commercial/production use until the licensor's change date and terms. Confirm the BUSL-1.1 grant permits your intended deployment (or relicense/replace those files) before shipping to production. The `SPDX-License-Identifier` at the top of each file is the authoritative license for that file.