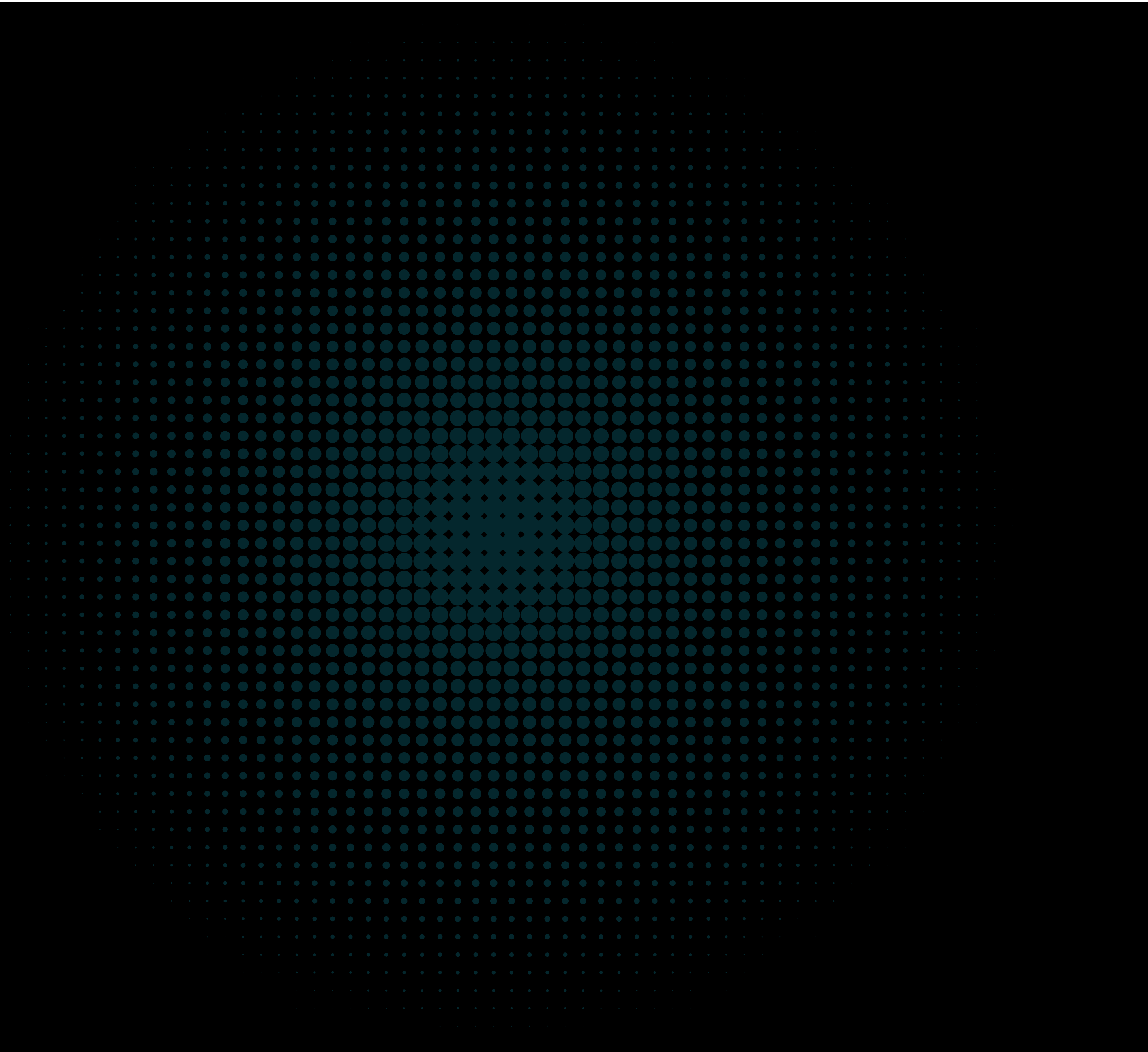
cmta.

CMTAT ACE Specification

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

CMTAT Chainlink ACE: v0.1.0
First published: May 2026



CMTAT ACE integration project

Introduction

This repository combines two components:

- **CMTAT (CMTA Token)**: an open security-token standard from the Capital Markets and Technology Association (CMTA), with compliance-oriented modules such as conditional transfer controls, account freeze/enforcement, token pause, document and snapshot engines, and token lifecycle controls.
- **Chainlink ACE (Automated Compliance Engine)**: a policy engine that evaluates configurable compliance and authorization policies at runtime for protected contract functions.

In this integration, CMTAT provides the token feature set and module structure, while ACE provides dynamic policy enforcement.
The goal is to let issuers update compliance behavior through policy configuration without changing core token business logic.

Table of Contents

- Deployment versions
- Changes from CMTAT
- TransferValidationPolicy
- ERC-165 Interface Support
- Library
- Initialize submodules
- Install dependencies
- Compile contracts
- Testing
- Linting & Formatting
- Scripts
- Audit Reports Summary
- Policy-Protected Functions (Current Integration)
- FAQ for Issuers Using CMTAT with ACE Policies

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 `PolicyProtectedUpgradeable` 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

Validation & Compliance

Aspect	CMTAT	Standard	Lite
Validation layer	`CMTATBaseRuleEngine` → `ValidationModuleRuleEngine`	`PolicyProtectedUpgradeable` → `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`

Initialization

The `Engine` struct parameter is replaced with `address policyEngine_`, `ISnapshotEngine snapshotEngine_`, and `IERC1643 documentEngine_`:

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

// ComplianceTokenCMTAT (Standard & Lite)
constructor(admin, ..., address policyEngine_, ISnapshotEngine snapshotEngine_,
IERC1643 documentEngine_)
```

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)`)
- SnapshotEngineModule, DocumentEngineModule
- ExtraInformationModule
- ERC20CrossChainModule, CCIPModule

Removed from Standard

- `CMTATBaseAccessControl` — replaced by `OwnableUpgradeable`
- `AccessControlModule` — role management removed from contract
- `CMTATBaseRuleEngine` — replaced by `PolicyProtectedUpgradeable`
- `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.

Removed from both variants

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

Added

- `PolicyProtectedUpgradeable` — 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](#transfervalidationpolicy) below)
- `ERC20TransferFromExtractor` — Extractor that produces 4 parameters (`spender`, `from`, `to`, `amount`) for `transfer()` and `transferFrom()`

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)`

Mock rules

Two 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`)
- **`RestrictedAddressRule`** — Rejects transfers involving addresses on a configurable restricted list (codes `14` / `15` for sender/recipient)

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()
}
```

ERC-165 Interface Support

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

- **Protected-token interface support**: `PolicyProtectedUpgradeable` 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.

Library

- CMTAT [v3.2.0]
- Chainlink ACE `1.0.0`
- 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:

```
npm install
```

Compile contracts

To compile

```
npx hardhat compile
```

Testing

To run tests:

```
npx hardhat test
```

Linting & Formatting

ESLint

Lint JavaScript files (tests, scripts, config):

```
npm run lint
```

Auto-fix fixable issues:

```
npm run lint:fix
```

Prettier

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

```
npm run format:check
```

Auto-format all files:

```
npm run format
```

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

Scripts

Deployment scripts

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)

Script	Description
`scripts/standard/deploy-standard-upgradeable.js`	Standard transparent proxy
`scripts/standard/deploy-standard-uups.js`	Standard UUPS proxy

Run any script with:

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

The script also configures RBAC operation allowances and grants roles (`MINTER_ROLE`, `BURNER_ROLE`, `BURNER_FROM_ROLE`, `ENFORCER_ROLE`, `ERC20ENFORCER_ROLE`, `DOCUMENT_ROLE`, `SNAPSHOOTER_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:

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

Audit Reports Summary

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

Slither

Here is the list of report performed with [Slither]

Setup:

```
python3 -m venv cct
chmod +x cct/bin/activate
source cct/bin/activate
pip install slither-analyzer
slither --version
```

Run:

```
source cct/bin/activate
npm run slither
```

```
slither . --checklist --filter-paths "openzeppelin-contracts|test|forge-std|mocks" > doc/audits/tools/slither-report.md
```

`npm 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/slither-report.md`.

When done, deactivate the virtual environment:

```
deactivate
```

Version	Report	Assessment
current	slither-report.md	slither-report-feedback.md

Report scope: repo-focused filtered checklist run.

0 High · 9 Medium · 10 Low · 27 Informational

ID	Finding	Instances	Assessment
M-1	`reentrancy-no-eth`	3	Contextual; expected external policy-engine calls and hook flow. Manual review required.
M-2	`uninitialized-local`	6	Likely analyzer limitation in extractor decode paths; treated as likely false positive.
L-1	`calls-loop`	8	Accepted by design where policy/rule chains iterate; monitor gas/complexity.

ID	Finding	Instances	Assessment
L-2	`reentrancy-events`	2	Informational reentrancy/event-order signal; no confirmed exploitable issue from checklist alone.
I-1	`assembly`	2	Expected in storage-slot patterns; informational.
I-2	`dead-code`	2	Cleanup candidate; not a direct security issue.
I-3	`naming-convention`	23	Style-only informational findings.

Aderyn

Here is the list of report performed with [Aderyn]

```
aderyn -x mocks --output doc/audits/tools/aderyn-report.md
```

Version	Report	Assessment
current	[aderyn-report.md]	[aderyn-report-feedback.md]

Report scope: 17 Solidity files, 959 nSLOC.

2 High · 10 Low

ID	Finding	Instances	Assessment
H-1	Arbitrary `from` passed to `transferFrom`	1	Accepted in context — policy-gated flow; not treated as exploitable in this integration design.
H-2	Contract locks Ether without withdraw	2	Accepted false positive — token deployments are not intended as ETH custody contracts.
L-1	Centralization Risk	11	Accepted by design — privileged governance/control is intentional.
L-2	Unsafe ERC20 Operation	7	Accepted false positive — primarily selector/module-flow usage, not unsafe token transfer wrappers.
L-3	Unspecific Solidity Pragma	17	Accepted by design — version ranges are intentionally used in this codebase.
L-4	Literal Instead of Constant	2	Informational — optional quality improvement.
L-5	PUSH0 Opcode	17	Environment-dependent informational finding in this setup.

ID	Finding	Instances	Assessment
L-6	Empty Block	22	Accepted by design — authorization hook pattern.
L-7	Loop Contains `require`/`revert`	4	Accepted by design — atomic validation and explicit failure signaling.
L-8	Unused State Variable	1	False positive — `STORAGE_LOCATION` is used via inline assembly in `_getStorage()`.
L-9	Costly operations inside loop	2	Accepted — expected tradeoff in policy/rule iteration paths.
L-10	Unused Import	9	Partially fixed; remaining cases are intentional (artifact/NatSpec/doc reasons).

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`
`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`

Function signature	Selector
`setTokenId(string)`	`0xdcfd616f`
`setDocumentEngine(address)`	`0x33611079`
`setSnapshotEngine(address)`	`0xe236aabf`
`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.

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.