Trustful Agents

Smart Contract Documentation

Version 1.3

January 2026

Deployed on Base Sepolia (Chain ID: 84532)

Contents

1. Introduction

1.1 Overview

Trustful Agents is a decentralized trust layer for AI agents built on the ERC-8004 validation standard. The system enables providers to signal trustworthiness through USDC collateral deposits, published Terms & Conditions (T&C), and council-based dispute resolution.

When an agent fails to act according to its own published T&C, affected parties (claimants) can file claims to be reimbursed for damages or losses from the agent's collateral. Independent councils evaluate claims and determine appropriate reimbursement amounts.

1.2 Architecture

The protocol consists of six core contracts that work together:

Contract	Purpose
CollateralVault	Holds USDC collateral per agent with withdrawal grace period and claim-based locking
TermsRegistry	Stores T&C versions per agent with content hash verification
CouncilRegistry	Manages dispute resolution councils, their members, and timing parameters
ClaimsManager	Handles claim lifecycle: filing, evidence periods, voting, and resolution
RulingExecutor	Executes council rulings, transfers collateral, distributes deposits
TrustfulValidator	Issues ERC-8004 validations based on trust conditions

1.3 Design Principles

Claim Reimbursement: The system is designed to reimburse claimants for damages or losses caused by agents not acting according to their own T&C. Collateral is transferred to claimants when claims are approved.

Council Neutrality: Claimant deposits always go to voting council members regardless of the claim outcome (approved or rejected). This eliminates any financial incentive for councils to favor approving or rejecting claims.

Front-running Prevention: A 7-day grace period on withdrawals prevents providers from withdrawing collateral after becoming aware of a potential claim.

Partial Awards: Councils can approve claims for less than the full amount requested, allowing for proportional reimbursement based on actual damages.

2. CollateralVault

The CollateralVault contract holds USDC collateral for each ERC-8004 agent. Collateral serves as a guarantee to reimburse claimants who suffer damages from agents not acting according to their T&C.

2.1 Key Features

Grace Period Withdrawals: Withdrawals require a 7-day grace period to prevent front-running claims.

Claim-based Locking: When a claim is filed, the claimed amount is locked until resolution.

Reimbursement Transfers: Approved claims result in collateral being transferred to claimants.

2.2 Data Structures

CollateralAccount

Parameter	Type	Description
`balance`	`uint256`	Total USDC deposited for the agent
`lockedAmount`	`uint256`	Amount locked for pending claims
`withdrawalInitiatedAt`	`uint256`	Timestamp when withdrawal was initiated (0 if none)
`withdrawalAmount`	`uint256`	Amount requested for withdrawal

2.3 Functions

deposit

```
function deposit(uint256 agentId, uint256 amount) external
```

Deposit USDC collateral for an agent. Only the agent owner should deposit collateral for their own agents.

Access: Permissionless (caller must approve USDC first)

Parameter	Type	Description
`agentId`	`uint256`	The ERC-8004 token ID of the agent
`amount`	`uint256`	Amount of USDC to deposit (6 decimals)

initiateWithdrawal

```
function initiateWithdrawal(uint256 agentId, uint256 amount) external
```

Start the withdrawal process. The withdrawal can be executed after the grace period (7 days).

Access: Agent Owner only

Parameter	Type	Description
`agentId`	`uint256`	The ERC-8004 token ID of the agent
`amount`	`uint256`	Amount of USDC to withdraw

cancelWithdrawal

```
function cancelWithdrawal(uint256 agentId) external
```

Cancel a pending withdrawal request.

Access: Agent Owner only

executeWithdrawal

```
function executeWithdrawal(uint256 agentId) external
```

Complete the withdrawal after the grace period has elapsed.

Access: Agent Owner only

getAccount

```
function getAccount(uint256 agentId) external view returns (CollateralAccount memory)
```

Get the collateral account details for an agent.

Returns: CollateralAccount struct with balance, locked amount, and withdrawal state

getAvailableBalance

```
function getAvailableBalance(uint256 agentId) external view returns (uint256)
```

Get the available (unlocked) balance for an agent.

Returns: Available balance that can be withdrawn

canExecuteWithdrawal

```
function canExecuteWithdrawal(uint256 agentId) external view returns (bool)
```

Check if a pending withdrawal can be executed.

Returns: True if grace period has elapsed

lock / unlock / transfer (Internal)

These functions are called by ClaimsManager and RulingExecutor to manage collateral during the claim lifecycle. They are not directly callable by users.

2.4 Errors

Error	Description
`NotAgentOwner`	Caller is not the owner of the agent
`InsufficientBalance`	Requested amount exceeds available balance
`InsufficientUnlockedBalance`	Requested amount exceeds unlocked balance
`NoWithdrawalPending`	No withdrawal has been initiated
`WithdrawalAlreadyPending`	A withdrawal is already in progress
`GracePeriodNotElapsed`	Must wait for grace period to complete
`ZeroAmount`	Amount cannot be zero
`AgentNotFound`	Agent does not exist in the registry

2.5 Events

Event	Parameters
`Deposited`	agentId (indexed), depositor, amount
`WithdrawalInitiated`	agentId (indexed), amount, executeAfter
`WithdrawalCancelled`	agentId (indexed)
`WithdrawalExecuted`	agentId (indexed), amount, recipient
`CollateralLocked`	agentId (indexed), claimId (indexed), amount
`CollateralUnlocked`	agentId (indexed), claimId (indexed), amount
`CollateralTransferred`	agentId (indexed), claimId (indexed), recipient, amount

3. TermsRegistry

The TermsRegistry stores Terms & Conditions (T&C) versions for each agent. T&C content is stored off-chain (typically on IPFS) with the content hash committed on-chain for verification.

3.1 Key Features

Version History: Multiple versions can be registered, with full history preserved for historical claims.

Content Verification: On-chain content hash allows verification that T&C has not been modified.

Council Binding: Each T&C version specifies which council handles disputes.

3.2 Data Structures

TermsVersion

Parameter	Type	Description
`contentHash`	`bytes32`	keccak256 hash of the T&C document
`contentUri`	`string`	IPFS URI or URL to the full document
`councilId`	`bytes32`	Council that handles disputes for these terms
`registeredAt`	`uint256`	Block timestamp when registered
`active`	`bool`	Whether this is the current active version

3.3 Functions

registerTerms

```
function registerTerms(uint256 agentId, bytes32 contentHash, string calldata
contentUri, bytes32 councilId) external returns (uint256 version)
```

Register a new T&C version for an agent. The new version automatically becomes active.

Access: Agent Owner only

Parameter	Type	Description
`agentId`	`uint256`	The ERC-8004 token ID of the agent
`contentHash`	`bytes32`	keccak256 hash of the T&C document
`contentUri`	`string`	URI to the full T&C document (IPFS recommended)
`councilId`	`bytes32`	Council that will handle disputes

Returns: The new version number

updateTerms

```
function updateTerms(uint256 agentId, bytes32 contentHash, string calldata
contentUri, bytes32 councilId) external returns (uint256 version)
```

Convenience function to register new T&C. Previous terms remain stored for historical claims.

Access: Agent Owner only

getActiveTerms

```
function getActiveTerms(uint256 agentId) external view returns (TermsVersion
memory terms, uint256 version)
```

Get the currently active T&C for an agent.

Returns: Active TermsVersion struct and version number

getTermsVersion

```
function getTermsVersion(uint256 agentId, uint256 version) external view
returns (TermsVersion memory)
```

Get T&C by specific version number.

hasActiveTerms

```
function hasActiveTerms(uint256 agentId) external view returns (bool)
```

Check if an agent has active (non-invalidated) T&C.

getCouncilForAgent

```
function getCouncilForAgent(uint256 agentId) external view returns (bytes32
councilId)
```

Get the council ID associated with the agent's active terms.

verifyTermsHash

```
function verifyTermsHash(uint256 agentId, uint256 version, bytes32
contentHash) external view returns (bool)
```

Verify that a content hash matches a specific terms version.

3.4 Errors

Error	Description
`NotAgentOwner`	Caller is not the owner of the agent
`AgentNotFound`	Agent does not exist in the registry
`NoActiveTerms`	Agent has no active T&C registered

`VersionNotFound`	Requested version does not exist
`CouncilNotActive`	The specified council is not active
`InvalidContentHash`	Content hash cannot be zero
`InvalidContentUri`	Content URI cannot be empty

3.5 Events

Event	Parameters
`TermsRegistered`	agentId (indexed), version (indexed), contentHash, contentUri, councilId
`TermsActivated`	agentId (indexed), version (indexed)
`TermsDeactivated`	agentId (indexed), version (indexed)
`TermsInvalidated`	agentId (indexed), reason

4. CouncilRegistry

The CouncilRegistry manages dispute resolution councils. Each council has members who vote on claims, timing parameters for evidence and voting periods, and deposit requirements.

4.1 Key Features

Council Management: Create and configure councils with customizable parameters.

Member Management: Add/remove members with a maximum of 11 active members per council.

Deposit Calculation: Configurable deposit percentage that claimants must pay.

Agent Override: Optionally assign specific agents to different councils.

4.2 Data Structures

Council

Parameter	Type	Description
`councilId`	`bytes32`	Unique identifier for the council
`name`	`string`	Human-readable council name
`owner`	`address`	Council administrator address
`evidencePeriod`	`uint256`	Duration for evidence submission (seconds)
`votingPeriod`	`uint256`	Duration for council voting (seconds)
`depositBasisPoints`	`uint256`	Deposit percentage (100 = 1%)
`memberCount`	`uint256`	Total number of members
`activeMemberCount`	`uint256`	Number of currently active members
`totalClaimsHandled`	`uint256`	Historical claims processed
`pendingClaims`	`uint256`	Currently active claims
`active`	`bool`	Whether council is accepting new claims

CouncilMember

Parameter	Type	Description
`member`	`address`	Member's wallet address
`joinedAt`	`uint256`	Timestamp when member was added
`votesCount`	`uint256`	Total votes cast by this member
`active`	`bool`	Whether member is currently active

4.3 Functions

createCouncil

```
function createCouncil(string calldata name, uint256 evidencePeriod, uint256 votingPeriod, uint256 depositBasisPoints) external returns (bytes32 councilId)
```

Create a new council. The caller becomes the council owner.

Access: Permissionless

Parameter	Type	Description
`name`	`string`	Human-readable name for the council
`evidencePeriod`	`uint256`	Evidence submission duration in seconds
`votingPeriod`	`uint256`	Voting duration in seconds
`depositBasisPoints`	`uint256`	Deposit percentage (100 = 1%, max 10000)

Returns: The new council ID (keccak256 of name and creator)

addMember

```
function addMember(bytes32 councilId, address member) external
```

Add a new member to the council. Maximum 11 active members allowed.

Access: Council Owner only

removeMember

```
function removeMember(bytes32 councilId, address member) external
```

Deactivate a member from the council.

Access: Council Owner only

setAgentCouncil

```
function setAgentCouncil(uint256 agentId, bytes32 councilId) external
```

Override the council for a specific agent (instead of using the T&C default).

Access: Agent Owner only

calculateRequiredDeposit

```
function calculateRequiredDeposit(bytes32 councilId, uint256 claimAmount) external view returns (uint256)
```

Calculate the deposit required for a given claim amount.

Returns: Required deposit in USDC (6 decimals)

isActiveMember

```
function isActiveMember(bytes32 councilId, address member) external view
returns (bool)
```

Check if an address is an active member of the council.

getActiveMemberCount

```
function getActiveMemberCount(bytes32 councilId) external view returns
(uint256)
```

Get the number of active members in a council.

4.4 Errors

Error	Description
`CouncilNotFound`	Council ID does not exist
`CouncilNotActive`	Council is not accepting new claims
`NotCouncilOwner`	Caller is not the council owner
`MemberAlreadyExists`	Address is already a member
`MemberNotFound`	Address is not a member
`TooManyMembers`	Council has reached 11 member limit
`InvalidPeriod`	Evidence or voting period is invalid
`InvalidDepositRate`	Deposit basis points exceeds 10000

4.5 Events

Event	Parameters
`CouncilCreated`	councilId (indexed), owner (indexed), name
`CouncilActivated`	councilId (indexed)
`CouncilDeactivated`	councilId (indexed)
`MemberAdded`	councilId (indexed), member (indexed)
`MemberRemoved`	councilId (indexed), member (indexed)
`AgentCouncilSet`	agentId (indexed), councilId (indexed)

5. ClaimsManager

The ClaimsManager handles the complete lifecycle of claims from filing through resolution. It manages deposits, voting periods, and coordinates with other contracts.

5.1 Key Features

Claim Filing: Claimants deposit USDC and lock agent collateral.

Evidence Period: Off-chain evidence submission before voting begins.

Council Voting: Active council members vote to approve, reject, or abstain.

Partial Awards: Approved amount can be less than claimed amount.

Vote Changes: Members can change their vote during the voting period.

5.2 Claim Lifecycle

Status	Description
Filed	Claim submitted, deposit paid, collateral locked, evidence period begins
EvidenceClosed	Evidence period ended, voting period active
Approved	Majority voted to approve, awaiting execution
Rejected	Majority voted to reject, awaiting execution
Expired	No votes were cast, awaiting execution
Executed	Ruling has been executed, claim is complete

5.3 Data Structures

Claim

Parameter	Type	Description
`claimId`	`uint256`	Unique identifier for the claim
`agentId`	`uint256`	The agent being claimed against
`claimant`	`address`	Address that filed the claim
`claimedAmount`	`uint256`	Amount requested by claimant
`approvedAmount`	`uint256`	Amount approved by council (if any)
`paymentReceiptHash`	`bytes32`	Hash of payment proof
`councilId`	`bytes32`	Council handling this claim
`claimantDeposit`	`uint256`	Deposit paid by claimant

lockedCollateral	uint256	Agent collateral locked for claim
status	ClaimStatus	Current claim status
filedAt	uint256	Timestamp when claim was filed
evidenceDeadline	uint256	When evidence period ends
votingDeadline	uint256	When voting period ends
hadVotes	bool	Whether any votes were cast

5.4 Functions

fileClaim

```
function fileClaim(uint256 agentId, uint256 claimedAmount, bytes32
paymentReceiptHash) external returns (uint256 claimId)
```

File a new claim against an agent. Requires USDC approval for the deposit.

Access: Permissionless (caller becomes the claimant)

Parameter	Type	Description
agentId	uint256	The agent to claim against
claimedAmount	uint256	Amount being claimed (min 1 USDC, max 1B USDC)
paymentReceiptHash	bytes32	Hash of payment evidence

Returns: The new claim ID

castVote

```
function castVote(uint256 claimId, Vote vote, uint256 approvedAmount, string
calldata reasoning) external
```

Cast a vote on a claim during the voting period.

Access: Active Council Members only

Parameter	Type	Description
claimId	uint256	The claim to vote on
vote	Vote	Approve, Reject, or Abstain
approvedAmount	uint256	Amount to approve (only if vote is Approve)
reasoning	string	Explanation for the vote

changeVote

```
function changeVote(uint256 claimId, Vote newVote, uint256 newApprovedAmount,
string calldata newReasoning) external
```

Change a previously cast vote during the voting period.

Access: Council Members who have already voted

finalizeClaim

```
function finalizeClaim(uint256 claimId) external
```

Finalize a claim after the voting period ends. Calculates the median approved amount.

Access: Permissionless

getClaim

```
function getClaim(uint256 claimId) external view returns (Claim memory)
```

Get full details of a claim.

getVotes

```
function getVotes(uint256 claimId) external view returns (VoteRecord[]
memory)
```

Get all votes cast on a claim.

5.5 Errors

Error	Description
`ClaimNotFound`	Claim ID does not exist
`InsufficientClaimAmount`	Claim amount below 1 USDC minimum
`ExcessiveClaimAmount`	Claim amount above 1 billion USDC maximum
`NoActiveTerms`	Agent has no active T&C
`InsufficientDeposit`	Deposit less than required amount
`NotCouncilMember`	Caller is not an active council member
`VotingPeriodNotStarted`	Still in evidence period
`VotingPeriodEnded`	Voting period has closed
`AlreadyVoted`	Member has already cast a vote
`NotYetVoted`	Cannot change vote before casting one
`ApprovedAmountExceedsClaimed`	Cannot approve more than claimed
`CouncilHasNoActiveMembers`	Council has no active members

5.6 Events

Event	Parameters
`ClaimFiled`	claimId (indexed), agentId (indexed), claimant, claimedAmount, deposit, councilId
`VoteCast`	claimId (indexed), voter (indexed), vote, approvedAmount
`VoteChanged`	claimId (indexed), voter (indexed), oldVote, newVote, newApprovedAmount
`ClaimFinalized`	claimId (indexed), status, approvedAmount

6. RulingExecutor

The RulingExecutor executes finalized claim rulings. It coordinates collateral transfers to claimants and distributes deposits to council members who voted.

6.1 Deposit Distribution Rules

Claimant deposits always go to council members who actually voted on the claim, regardless of the outcome. This ensures councils have no financial incentive to approve or reject claims.

Outcome	Collateral	Deposit
Approved	Transferred to claimant	Distributed to voters
Rejected	Unlocked (stays with agent)	Distributed to voters
Expired (with votes)	Unlocked (stays with agent)	Distributed to voters
Expired (no votes)	Unlocked (stays with agent)	Returned to claimant

6.2 Functions

executeApprovedClaim

```
function executeApprovedClaim(uint256 claimId) external
```

Execute an approved claim. Transfers collateral to claimant and distributes deposit to voters.

Access: Permissionless

executeRejectedClaim

```
function executeRejectedClaim(uint256 claimId) external
```

Execute a rejected claim. Unlocks collateral and distributes deposit to voters.

Access: Permissionless

executeExpiredClaim

```
function executeExpiredClaim(uint256 claimId) external
```

Execute an expired claim. Returns deposit to claimant only if no votes were cast.

Access: Permissionless

6.3 Errors

Error	Description
`ClaimNotFound`	Claim ID does not exist
`ClaimNotFinalized`	Claim has not been finalized yet

`ClaimAlreadyExecuted`	Claim has already been executed
`InvalidClaimStatus`	Claim status does not match function

6.4 Events

Event	Parameters
`ClaimExecuted`	claimId (indexed), agentId (indexed), claimant, payoutAmount, returnedDeposit
`DepositDistributed`	claimId (indexed), recipient (indexed), amount

7. TrustfulValidator

The TrustfulValidator issues ERC-8004 validation NFTs to agents that meet trust conditions. These validations signal to clients that an agent has deposited collateral and published T&C.

7.1 Trust Conditions

An agent must meet the following conditions to receive validation:

1. Have active Terms & Conditions registered in TermsRegistry
2. Have at least the minimum required collateral deposited
3. Have the collateral available (not fully locked by claims)

7.2 Functions

requestValidation

```
function requestValidation(uint256 agentId) external returns (uint256 validationId)
```

Request an ERC-8004 validation for an agent. Validation is issued if trust conditions are met.

Access: Agent Owner only

Returns: The validation token ID

revokeValidation

```
function revokeValidation(uint256 agentId) external
```

Revoke an existing validation. Called automatically when trust conditions are no longer met.

Access: Agent Owner or Governance

getTrustInfo

```
function getTrustInfo(uint256 agentId) external view returns (TrustInfo memory)
```

Get comprehensive trust information for an agent.

TrustInfo Structure

Parameter	Type	Description
`isValid`	`bool`	Whether agent has active validation
`hasTerms`	`bool`	Whether agent has registered T&C
`totalCollateral`	`uint256`	Total deposited collateral
`availableCollateral`	`uint256`	Collateral not locked by claims
`lockedCollateral`	`uint256`	Collateral locked for pending claims

activeClaimsCount	uint256	Number of pending claims
termsHash	bytes32	Hash of current T&C
termsUri	string	URI to current T&C document
councilId	bytes32	Council handling disputes

checkTrustConditions

```
function checkTrustConditions(uint256 agentId) external view returns (bool meetsConditions, string memory reason)
```

Check if an agent meets all trust conditions for validation.

7.3 Events

Event	Parameters
ValidationIssued	agentId (indexed), validationId (indexed)
ValidationRevoked	agentId (indexed), validationId (indexed), reason

8. Supporting Contracts

8.1 TrustfulPausable

Base contract providing emergency pause functionality with granular scope control.

Pause Scopes

Scope	Affected Operations
All	Pauses all contract operations
Deposits	Pauses collateral deposits
Withdrawals	Pauses collateral withdrawals
Claims	Pauses new claim filing
Voting	Pauses vote casting and changes
Executions	Pauses ruling execution

8.2 GovernanceMultisig

2-of-3 multisig contract controlling governance operations across all protocol contracts.

Governance can: pause contracts, update contract references, set minimum collateral requirements, and execute emergency actions.

9. Integration Guide

9.1 For Providers (Agent Operators)

Providers operate AI agents and want to signal trustworthiness to potential clients.

Step 1: Deposit Collateral

After minting your ERC-8004 agent, deposit USDC collateral to back your agent.

```
// First, approve the CollateralVault to spend your USDC
IERC20 usdc = IERC20(USDC_ADDRESS);
usdc.approve(address(collateralVault), 10_000e6); // 10,000 USDC

// Then deposit collateral for your agent
uint256 agentId = 42; // Your agent's token ID
collateralVault.deposit(agentId, 10_000e6);
```

Step 2: Register Terms & Conditions

Upload your T&C document to IPFS and register it on-chain.

```
// Compute the content hash of your T&C document
bytes32 contentHash = keccak256(abi.encodePacked(tcDocumentContent));

// IPFS URI where the document is stored
string memory contentUri = "ipfs://QmYourContentHash...";

// The council that will handle disputes (get from CouncilRegistry)
bytes32 councilId = 0x1234...;

// Register the terms
uint256 version = termsRegistry.registerTerms(
    agentId,
    contentHash,
    contentUri,
    councilId
);
```

Step 3: Request Validation

Once collateral is deposited and T&C registered, request an ERC-8004 validation.

```
// Request validation (will revert if conditions not met)
uint256 validationId = trustfulValidator.requestValidation(agentId);

// You can check if conditions are met first
(bool meetsConditions, string memory reason) =
    trustfulValidator.checkTrustConditions(agentId);
require(meetsConditions, reason);
```

Withdrawing Collateral

To withdraw collateral, initiate a withdrawal and wait for the grace period.

```
// Step 1: Initiate withdrawal (starts 7-day grace period)
collateralVault.initiateWithdrawal(agentId, 5_000e6); // Withdraw 5,000 USDC

// Step 2: Wait 7 days...

// Step 3: Execute withdrawal after grace period
if (collateralVault.canExecuteWithdrawal(agentId)) {
    collateralVault.executeWithdrawal(agentId);
}

// Optional: Cancel withdrawal if needed
collateralVault.cancelWithdrawal(agentId);
```

9.2 For Clients (Agent Users)

Clients use AI agents and want to verify their trustworthiness before engaging.

Verify Agent Trust Status

Before using an agent, check its trust information.

```
uint256 agentId = 42; // The agent you want to verify

// Get comprehensive trust info
ITrustfulValidator.TrustInfo memory info =
    trustfulValidator.getTrustInfo(agentId);

// Check if agent is validated
require(info.isValid, "Agent not validated");
```

```
// Check available collateral meets your risk tolerance
uint256 minCollateral = 1_000e6; // Require at least 1,000 USDC
require(info.availableCollateral >= minCollateral, "Insufficient
collateral");

// Fetch and review T&C from the URI
string memory termsUri = info.termsUri;
// Fetch document from termsUri (e.g., IPFS gateway)
// Verify hash matches: keccak256(document) == info.termsHash
```

Check Specific Conditions

```
// Check if agent has active terms
bool hasTerms = termsRegistry.hasActiveTerms(agentId);

// Get collateral details
ICollateralVault.CollateralAccount memory account =
    collateralVault.getAccount(agentId);
uint256 totalBalance = account.balance;
uint256 lockedAmount = account.lockedAmount;
uint256 available = collateralVault.getAvailableBalance(agentId);

// Check pending claims count
IClaimsManager.ClaimStats memory stats =
    claimsManager.getAgentStats(agentId);
uint256 pendingClaims = stats.pendingClaims;
```

9.3 For Claimants

Claimants have suffered damages from an agent not following its T&C.

Step 1: Calculate Required Deposit

Before filing, calculate how much deposit you need to pay.

```
uint256 agentId = 42; // The agent you're claiming against
uint256 claimAmount = 500e6; // Claiming 500 USDC in damages

// Get the council that handles this agent's disputes
bytes32 councilId = termsRegistry.getCouncilForAgent(agentId);
```

```
// Calculate required deposit
uint256 requiredDeposit = councilRegistry.calculateRequiredDeposit(
    councilId,
    claimAmount
);
// e.g., if deposit rate is 5%, requiredDeposit = 25 USDC
```

Step 2: File the Claim

Approve USDC for the deposit and file your claim.

```
// Prepare payment receipt hash (evidence of your payment to the agent)
bytes32 paymentReceiptHash = keccak256(abi.encodePacked(
    "tx:0xabc123...", // Transaction hash
    block.timestamp
));

// Approve deposit
usdc.approve(address(claimsManager), requiredDeposit);

// File the claim
uint256 claimId = claimsManager.fileClaim(
    agentId,
    claimAmount,
    paymentReceiptHash
);
```

Step 3: Monitor Claim Status

Track your claim through the resolution process.

```
// Get claim details
IClaimsManager.Claim memory claim = claimsManager.getClaim(claimId);

// Check current status
IClaimsManager.ClaimStatus status = claim.status;
// Filed -> EvidenceClosed -> Approved/Rejected/Expired -> Executed

// Check deadlines
uint256 evidenceDeadline = claim.evidenceDeadline;
```

```
uint256 votingDeadline = claim.votingDeadline;

// Get votes on your claim
IClaimsManager.VoteRecord[] memory votes = claimsManager.getVotes(claimId);
```

Step 4: Finalize and Execute

After voting ends, anyone can finalize and execute the claim.

```
// Finalize the claim (calculates outcome based on votes)
// Can be called by anyone after voting deadline
claimsManager.finalizeClaim(claimId);

// Check the outcome
claim = claimsManager.getClaim(claimId);

// Execute based on status
if (claim.status == IClaimsManager.ClaimStatus.Approved) {
    // You won! Execute to receive reimbursement
    rulingExecutor.executeApprovedClaim(claimId);
    // claim.approvedAmount will be transferred to you
} else if (claim.status == IClaimsManager.ClaimStatus.Rejected) {
    rulingExecutor.executeRejectedClaim(claimId);
    // Collateral unlocked, deposit goes to voters
} else if (claim.status == IClaimsManager.ClaimStatus.Expired) {
    rulingExecutor.executeExpiredClaim(claimId);
    // If no votes: deposit returned to you
}
```

9.4 For Council Members

Council members review claims and vote on outcomes.

Query Pending Claims

Find claims assigned to your council that need attention.

```
bytes32 councilId = 0x1234...; // Your council ID

// Get all claim IDs for this council
uint256[] memory claimIds = claimsManager.getClaimsByCouncil(councilId);
```

```
// Filter for claims in voting period
for (uint256 i = 0; i < claimIds.length; i++) {
    IClaimsManager.Claim memory claim = claimsManager.getClaim(claimIds[i]);

    // Check if in voting period
    bool inVotingPeriod =
        block.timestamp > claim.evidenceDeadline &&
        block.timestamp <= claim.votingDeadline;

    if (inVotingPeriod && claim.status !=
IClaimsManager.ClaimStatus.Executed) {
        // This claim needs your vote
    }
}
```

Cast Your Vote

Vote on a claim during the voting period.

```
uint256 claimId = 123;

// Option 1: Approve the full claimed amount
claimsManager.castVote(
    claimId,
    IClaimsManager.Vote.Approve,
    claim.claimedAmount, // Full amount
    "Evidence clearly shows T&C violation"
);

// Option 2: Approve a partial amount
claimsManager.castVote(
    claimId,
    IClaimsManager.Vote.Approve,
    250e6, // Only 250 USDC of the 500 USDC claimed
    "Partial damages verified, rest not substantiated"
);

// Option 3: Reject the claim
claimsManager.castVote(
```

```
    claimId,
    IClaimsManager.Vote.Reject,
    0, // Amount ignored for rejections
    "No evidence of T&C violation"
);

// Option 4: Abstain
claimsManager.castVote(
    claimId,
    IClaimsManager.Vote.Abstain,
    0,
    "Conflict of interest"
);
```

Change Your Vote

You can change your vote anytime during the voting period.

```
// Changed your mind after reviewing more evidence
claimsManager.changeVote(
    claimId,
    IClaimsManager.Vote.Approve,
    400e6, // New amount
    "Additional evidence submitted, updating assessment"
);
```

Check Your Voting Stats

```
// Get your member info
ICouncilRegistry.CouncilMember memory myInfo =
    councilRegistry.getMember(councilId, myAddress);

uint256 totalVotesCast = myInfo.votesCount;
bool isActive = myInfo.active;
```

Appendix A: Function Reference

A.1 Governance-Only Functions

These functions can only be called by the governance multisig.

Function	Description
`CollateralVault.setClaimsManager`	Set the ClaimsManager contract address
`CollateralVault.setRulingExecutor`	Set the RulingExecutor contract address
`ClaimsManager.setCollateralVault`	Set the CollateralVault contract address
`ClaimsManager.setTermsRegistry`	Set the TermsRegistry contract address
`ClaimsManager.setCouncilRegistry`	Set the CouncilRegistry contract address
`ClaimsManager.setRulingExecutor`	Set the RulingExecutor contract address
`RulingExecutor.setClaimsManager`	Set the ClaimsManager contract address
`RulingExecutor.setCollateralVault`	Set the CollateralVault contract address
`RulingExecutor.setCouncilRegistry`	Set the CouncilRegistry contract address
`TrustfulPausable.pause`	Pause contract operations by scope
`TrustfulPausable.unpause`	Unpause contract operations by scope

A.2 Owner-Restricted Functions

These functions require the caller to be the owner of the relevant entity.

Function	Description
`CollateralVault.initiateWithdrawal`	Agent owner initiates withdrawal
`CollateralVault.cancelWithdrawal`	Agent owner cancels withdrawal
`CollateralVault.executeWithdrawal`	Agent owner executes withdrawal
`TermsRegistry.registerTerms`	Agent owner registers T&C
`TermsRegistry.updateTerms`	Agent owner updates T&C
`CouncilRegistry.addMember`	Council owner adds member
`CouncilRegistry.removeMember`	Council owner removes member
`CouncilRegistry.updateCouncil`	Council owner updates parameters
`CouncilRegistry.setAgentCouncil`	Agent owner overrides council
`TrustfulValidator.requestValidation`	Agent owner requests validation

A.3 Council Member Functions

These functions can only be called by active council members.

Function	Description
`ClaimsManager.castVote`	Cast a vote on a claim
`ClaimsManager.changeVote`	Change a previously cast vote

A.4 Permissionless Functions

These functions can be called by anyone.

Function	Description
`CollateralVault.deposit`	Deposit collateral for an agent
`CouncilRegistry.createCouncil`	Create a new council
`ClaimsManager.fileClaim`	File a claim against an agent
`ClaimsManager.finalizeClaim`	Finalize a claim after voting
`RulingExecutor.executeApprovedClaim`	Execute an approved claim
`RulingExecutor.executeRejectedClaim`	Execute a rejected claim
`RulingExecutor.executeExpiredClaim`	Execute an expired claim
`All view functions`	Query contract state

A.5 Version History

Version	Changes
v1.0	Initial release with core contracts
v1.1	Added TrustfulValidator and complete architecture
v1.2	Vote changing, council closure, deposit distribution to voters only
v1.3	Audit fixes: removed on-chain evidence, underflow protection, MAX_CLAIM_AMOUNT, council member limits