```
"""
ORACLE_CACHE V2: Production-Grade Governed Distributed Cache
Grimoire Codex Implementation with Full Governance Layer

OPERATOR: FINALIZE

Complete system including:
- Phase 1: Cache invalidation (Chronom, Relata, Echo, Hydrina, Byzantium, Entangla, Oedipha,
Deadpoolia)
- Phase 2: Governance layer (Logora, Revela, Ma'atara, Moirae, Zephyrus, Heraia)
"""

import time
import threading
import hashlib
import json
from typing import Dict, List, Set, Optional, Callable, Any, Tuple
from dataclasses import dataclass, field
from enum import Enum
from collections import defaultdict
from datetime import datetime

#============================================================================
=
# ORIGIN: Root Rune Configuration
#============================================================================
=

class FacetConfig:
    """Enabled Facets: Time, Network, State, Recovery, Prediction, Consensus, Governance"""
    TEMPORAL = True
    NETWORK = True
    STATE = True
    RECOVERY = True
    PREDICTION = True
    CONSENSUS = True
    GOVERNANCE = True  # NEW

#============================================================================
=
# PHASE 1: CORE CACHE INVALIDATION SPELLS
#============================================================================
=
@dataclass
class TemporalSnapshot:
    version: int
    timestamp: float
    data: Any
    dependencies: Set[str] = field(default_factory=set)

class Chronom:
    """Temporal snapshots and version control for cache entries"""

    def __init__(self):
        self.versions: Dict[str, List[TemporalSnapshot]] = defaultdict(list)
        self.current_version: Dict[str, int] = {}
        self.lock = threading.Lock()

    def snapshot(self, key: str, data: Any, deps: Set[str] = None):
        """Create temporal snapshot of cache entry"""
        with self.lock:
            version = self.current_version.get(key, 0) + 1
            self.current_version[key] = version

            snap = TemporalSnapshot(
                version=version,
                timestamp=time.time(),
                data=data,
                dependencies=deps or set()
            )
            self.versions[key].append(snap)
            return version

    def rollback(self, key: str, version: int) -> Optional[Any]:
        """Rollback to specific version"""
        with self.lock:
            if key in self.versions:
                for snap in reversed(self.versions[key]):
                    if snap.version == version:
                        return snap.data
            return None

    def get_dependencies(self, key: str, version: int) -> Set[str]:
        """Get dependency set for specific version"""
        with self.lock:
            if key in self.versions:
                for snap in reversed(self.versions[key]):
```

```
                if snap.version == version:
                    return snap.dependencies
            return set()

class Relata:
    """Dependency and relationship graph for cache entries"""

    def __init__(self):
        self.forward_deps: Dict[str, Set[str]] = defaultdict(set)
        self.reverse_deps: Dict[str, Set[str]] = defaultdict(set)
        self.lock = threading.Lock()

    def add_dependency(self, key: str, depends_on: str):
        """Add dependency relationship"""
        with self.lock:
            self.forward_deps[key].add(depends_on)
            self.reverse_deps[depends_on].add(key)

    def get_cascade_targets(self, key: str) -> List[str]:
        """Get all keys that must be invalidated when key is invalidated"""
        with self.lock:
            visited = set()
            queue = [key]
            cascade = []

            while queue:
                current = queue.pop(0)
                if current in visited:
                    continue

                visited.add(current)
                cascade.append(current)

                for dependent in self.reverse_deps.get(current, set()):
                    if dependent not in visited:
                        queue.append(dependent)

            return cascade[1:]

@dataclass
class InvalidationEvent:
    key: str
    timestamp: float
    cascade_targets: List[str]
    reason: str

class Echo:
    """System-wide broadcast of invalidation events"""

    def __init__(self):
        self.subscribers: List[Callable[[InvalidationEvent], None]] = []
        self.event_log: List[InvalidationEvent] = []
        self.lock = threading.Lock()

    def subscribe(self, callback: Callable[[InvalidationEvent], None]):
        """Subscribe to invalidation events"""
        with self.lock:
            self.subscribers.append(callback)

    def broadcast(self, event: InvalidationEvent):
        """Broadcast invalidation event to all subscribers"""
        with self.lock:
            self.event_log.append(event)
            for callback in self.subscribers:
                threading.Thread(target=callback, args=(event,)).start()

class Hydrina:
    """Auto-spawn fresh cache entries on invalidation"""

    def __init__(self):
        self.regenerators: Dict[str, Callable[[], Any]] = {}
        self.lock = threading.Lock()

    def register_regenerator(self, key: str, regenerator: Callable[[], Any]):
        """Register function to regenerate cache entry"""
        with self.lock:
            self.regenerators[key] = regenerator

    def regenerate(self, key: str) -> Optional[Any]:
        """Regenerate cache entry"""
        with self.lock:
            if key in self.regenerators:
                return self.regenerators[key]()
            return None

    def has_regenerator(self, key: str) -> bool:
        """Check if key has a regenerator"""
        with self.lock:
```

```
        return key in self.regenerators

class ConsensusVote:
    VALID = "valid"
    STALE = "stale"
    UNKNOWN = "unknown"

class Byzantium:
    """Distributed consensus for cache validity"""

    def __init__(self, node_id: str, quorum_size: int = 3):
        self.node_id = node_id
        self.quorum_size = quorum_size
        self.votes: Dict[str, Dict[str, str]] = defaultdict(dict)
        self.lock = threading.Lock()

    def vote(self, key: str, status: str, node_id: str = None):
        """Cast vote on cache validity"""
        node = node_id or self.node_id
        with self.lock:
            self.votes[key][node] = status

    def get_consensus(self, key: str) -> Optional[str]:
        """Get consensus status for key"""
        with self.lock:
            if key not in self.votes or len(self.votes[key]) < self.quorum_size:
                return None

            vote_counts = defaultdict(int)
            for vote in self.votes[key].values():
                vote_counts[vote] += 1

            for status, count in vote_counts.items():
                if count >= (self.quorum_size // 2 + 1):
                    return status

            return ConsensusVote.UNKNOWN

class Entangla:
    """Correlated state syncing across distributed caches"""

    def __init__(self):
        self.entangled_pairs: Dict[str, Set[str]] = defaultdict(set)
        self.lock = threading.Lock()
    def entangle(self, key1: str, key2: str):
        """Create entanglement between two cache keys"""
        with self.lock:
            self.entangled_pairs[key1].add(key2)
            self.entangled_pairs[key2].add(key1)

    def get_entangled(self, key: str) -> Set[str]:
        """Get all keys entangled with this key"""
        with self.lock:
            return self.entangled_pairs.get(key, set()).copy()

class Oedipha:
    """Predictive analytics for cache staleness"""

    def __init__(self):
        self.access_patterns: Dict[str, List[float]] = defaultdict(list)
        self.update_patterns: Dict[str, List[float]] = defaultdict(list)
        self.lock = threading.Lock()

    def record_access(self, key: str):
        """Record cache access"""
        with self.lock:
            self.access_patterns[key].append(time.time())

    def record_update(self, key: str):
        """Record cache update"""
        with self.lock:
            self.update_patterns[key].append(time.time())

    def predict_staleness_time(self, key: str) -> Optional[float]:
        """Predict when cache will become stale"""
        with self.lock:
            # Require at least 3 updates to make a reliable prediction
            if key not in self.update_patterns or len(self.update_patterns[key]) < 3:
                return None

            updates = self.update_patterns[key]
            intervals = [updates[i+1] - updates[i] for i in range(len(updates)-1)]
            avg_interval = sum(intervals) / len(intervals)

            return updates[-1] + avg_interval

class RegenerationState(Enum):
    """State machine for regeneration process"""
    VALID = "valid"
    INVALIDATED = "invalidated"
    BLOCKED = "blocked"
    REGENERATING = "regenerating"
    READY = "ready"

@dataclass
class RegenerationPhase:
    """A phase in the regeneration cascade"""
    phase_number: int
    keys: List[str]
    dependencies_satisfied: bool = False

class Deadpoolia:
    """Dependency-ordered regeneration cascade"""

    def __init__(self, relata: Relata):
        self.relata = relata
        self.state: Dict[str, RegenerationState] = {}
        self.lock = threading.Lock()
        self.cycle_detected = False
        self.cycle_path: List[str] = []

    def topological_sort(self, keys: Set[str]) -> List[List[str]]:
        """Compute regeneration phases via topological sort"""
        in_degree = {k: 0 for k in keys}
        for key in keys:
            deps = self.relata.forward_deps.get(key, set())
            in_degree[key] = len(deps & keys)

        phases = []
        remaining = set(keys)

        while remaining:
            phase = [k for k in remaining if in_degree[k] == 0]

            if not phase:
                self.cycle_detected = True
                self.cycle_path = list(remaining)
                raise ValueError(f"Circular dependency detected in: {remaining}")

            phases.append(phase)
            for key in phase:
                remaining.remove(key)
                for dependent in self.relata.reverse_deps.get(key, set()):
                    if dependent in remaining:
                        in_degree[dependent] -= 1

        return phases

    def assess_damage(self, invalidated_keys: Set[str]) -> List[RegenerationPhase]:
        """Assess damage and build regeneration plan"""
        with self.lock:
            for key in invalidated_keys:
                self.state[key] = RegenerationState.INVALIDATED

        try:
            phase_keys = self.topological_sort(invalidated_keys)

            phases = []
            for i, keys in enumerate(phase_keys):
                phases.append(RegenerationPhase(
                    phase_number=i,
                    keys=keys,
                    dependencies_satisfied=(i == 0)
                ))

            return phases

        except ValueError as e:
            print(f"⚠️  DEADPOOLIA: {e}")
            return []

    def check_dependencies(self, key: str, invalidated_set: Set[str]) -> bool:
        """Check if all dependencies for a key are satisfied"""
        with self.lock:
            deps = self.relata.forward_deps.get(key, set())
            relevant_deps = deps & invalidated_set

            for dep in relevant_deps:
                dep_state = self.state.get(dep, RegenerationState.VALID)
                if dep_state not in [RegenerationState.READY, RegenerationState.VALID]:
                    return False

            return True
    def mark_ready(self, key: str):
        """Mark key as ready after successful regeneration"""
        with self.lock:
            self.state[key] = RegenerationState.READY

    def get_state(self, key: str) -> RegenerationState:
        """Get current regeneration state of key"""
        with self.lock:
            return self.state.get(key, RegenerationState.VALID)

#============================================================================
=
# PHASE 2: GOVERNANCE LAYER
#============================================================================
=

# LOGORA: Logos - Language as Creation (Creation Attribution)
#============================================================================
=

@dataclass
class CreationRecord:
    """What spell created you?"""
    spell_composition: str          # e.g. "EMERGE(Chronom + Hydrina)"
    operator_chain: List[str]       # e.g. ["CHAIN", "NEST", "EMERGE"]
    creator_id: str                 # Who invoked the creation
    creation_timestamp: float
    creation_context: Dict[str, Any] = field(default_factory=dict)

    def __str__(self):
        return f"Created by {self.creator_id} using {self.spell_composition} at
{datetime.fromtimestamp(self.creation_timestamp)}"

class Logora:
    """
    OPERATOR: WRAP
    Creation attribution system - tracks "What spell created you?"
    """

    def __init__(self):
        self.creation_records: Dict[str, CreationRecord] = {}
        self.lock = threading.Lock()

    def record_creation(self, key: str, spell: str, operators: List[str],
                creator_id: str, context: Dict = None):
        """Record the creation of a cache entry"""
        with self.lock:
            self.creation_records[key] = CreationRecord(
                spell_composition=spell,
                operator_chain=operators,
                creator_id=creator_id,
                creation_timestamp=time.time(),
                creation_context=context or {}
            )

    def get_genesis(self, key: str) -> Optional[CreationRecord]:
        """Get creation record for a key"""
        with self.lock:
            return self.creation_records.get(key)

    def who_created(self, key: str) -> Optional[str]:
        """Get creator ID for a key"""
        record = self.get_genesis(key)
        return record.creator_id if record else None

# REVELA: Hidden Knowledge - Modification Provenance
#============================================================================
=

@dataclass
class ModificationRecord:
    """What modified you?"""
    timestamp: float
    spell_used: str
    operator_used: str
    modifier_id: str
    change_reason: str
    previous_value_hash: str
    new_value_hash: str

    def __str__(self):
        return f"{self.modifier_id} modified via {self.spell_used} ({self.change_reason}) at
{datetime.fromtimestamp(self.timestamp)}"

class Revela:
    """
    OPERATOR: CHAIN
    Modification provenance - tracks "What cloth modified you?"
    """

    def __init__(self):
        self.modification_logs: Dict[str, List[ModificationRecord]] = defaultdict(list)
        self.lock = threading.Lock()

    def record_modification(self, key: str, spell: str, operator: str,
                            modifier_id: str, reason: str,
                            old_value: Any, new_value: Any):
        """Record a modification to a cache entry"""
        with self.lock:
            record = ModificationRecord(
                timestamp=time.time(),
                spell_used=spell,
                operator_used=operator,
                modifier_id=modifier_id,
                change_reason=reason,
                previous_value_hash=self._hash_value(old_value),
                new_value_hash=self._hash_value(new_value)
            )
            self.modification_logs[key].append(record)

    def get_provenance(self, key: str) -> List[ModificationRecord]:
        """Get full modification history for a key"""
        with self.lock:
            return list(self.modification_logs.get(key, []))

    def get_last_modifier(self, key: str) -> Optional[str]:
        """Get the last person who modified this key"""
        with self.lock:
            logs = self.modification_logs.get(key, [])
            return logs[-1].modifier_id if logs else None

    def _hash_value(self, value: Any) -> str:
        """Hash a value for audit trail"""
        value_str = json.dumps(value, sort_keys=True, default=str)
        return hashlib.sha256(value_str.encode()).hexdigest()[:16]

# MA'ATARA: Ma'at - Order and Justice (Policy Enforcement)
#============================================================================
=

@dataclass
class PolicyRule:
    """A single policy rule"""
    name: str
    condition: Callable[[Any], bool]
    enforcement_level: str  # "STRICT", "WARN", "AUDIT"
    violation_action: str   # "REJECT", "LOG", "ALERT"
    description: str = ""

@dataclass
class PolicyBinding:
    """What law manual approved you?"""
    rules: List[PolicyRule] = field(default_factory=list)
    compliance_status: str = "COMPLIANT"
    last_audit: float = 0.0
    violations: List[str] = field(default_factory=list)

class Ma_atara:
    """
    OPERATOR: LAYER
    Policy enforcement engine - "What law manual approved you?"
    """

    def __init__(self):
        self.global_policies: List[PolicyRule] = []
        self.key_policies: Dict[str, List[PolicyRule]] = defaultdict(list)
        self.violation_log: List[Tuple[str, str, float]] = []
        self.lock = threading.Lock()

    def add_global_policy(self, rule: PolicyRule):
        """Add a policy that applies to all cache entries"""
        with self.lock:
            self.global_policies.append(rule)

    def add_key_policy(self, key: str, rule: PolicyRule):
        """Add a policy specific to a key"""
        with self.lock:
            self.key_policies[key].append(rule)

    def validate(self, key: str, value: Any) -> Tuple[bool, List[str]]:
        """
        Validate a value against all applicable policies
        Returns: (is_valid, list_of_violations)
        """
        with self.lock:
            violations = []
            # Check global policies
            for rule in self.global_policies:
                if not rule.condition(value):
                    violation = f"Policy '{rule.name}' violated"
                    violations.append(violation)

                    if rule.enforcement_level == "STRICT":
                        self._log_violation(key, violation)
                        return False, violations

            # Check key-specific policies
            for rule in self.key_policies.get(key, []):
                if not rule.condition(value):
                    violation = f"Policy '{rule.name}' violated"
                    violations.append(violation)

                    if rule.enforcement_level == "STRICT":
                        self._log_violation(key, violation)
                        return False, violations

            return len(violations) == 0, violations

    def _log_violation(self, key: str, violation: str):
        """Log a policy violation"""
        self.violation_log.append((key, violation, time.time()))

    def get_violations(self, key: str = None) -> List[Tuple[str, str, float]]:
        """Get policy violations, optionally filtered by key"""
        with self.lock:
            if key:
                return [(k, v, t) for k, v, t in self.violation_log if k == key]
            return list(self.violation_log)

# MOIRAE: Life Thread - Lifecycle Manager
#=============================================================================
=

@dataclass
class LifecycleContract:
    """What would make you obsolete?"""
    max_age_seconds: Optional[float] = None
    expiration_conditions: List[Callable[[], bool]] = field(default_factory=list)
    obsolescence_triggers: Set[str] = field(default_factory=set)
    auto_cleanup: bool = True
    death_callback: Optional[Callable[[], None]] = None
    birth_timestamp: float = field(default_factory=time.time)

class Moirae:
    """
    OPERATOR: NEST
    Lifecycle management - "What would make you obsolete?"
    The Fates who cut the thread of life
    """

    def __init__(self):
        self.contracts: Dict[str, LifecycleContract] = {}
        self.expiration_queue: List[Tuple[str, float]] = []  # (key, expiration_time)
        self.lock = threading.Lock()
        self.cleanup_thread = None
        self.running = False

    def bind_lifecycle(self, key: str, contract: LifecycleContract):
        """Bind a lifecycle contract to a key"""
        with self.lock:
            contract.birth_timestamp = time.time()
            self.contracts[key] = contract

            # Add to expiration queue if has max_age
            if contract.max_age_seconds:
                expiration_time = contract.birth_timestamp + contract.max_age_seconds
                self.expiration_queue.append((key, expiration_time))
                self.expiration_queue.sort(key=lambda x: x[1])

    def is_expired(self, key: str) -> Tuple[bool, Optional[str]]:
        """
        Check if a key has expired
        Returns: (is_expired, reason)
        """
        with self.lock:
            contract = self.contracts.get(key)
            if not contract:
                return False, None

            # Check age-based expiration
            if contract.max_age_seconds:
                age = time.time() - contract.birth_timestamp
                if age > contract.max_age_seconds:
```

```
                return True, f"max_age exceeded ({age:.2f}s > {contract.max_age_seconds}s)"

        # Check condition-based expiration
        for i, condition in enumerate(contract.expiration_conditions):
            try:
                if condition():
                    return True, f"expiration_condition_{i} triggered"
            except Exception as e:
                print(f"⚠️ Error checking expiration condition: {e}")

        return False, None

    def get_contract(self, key: str) -> Optional[LifecycleContract]:
        """Get lifecycle contract for a key"""
        with self.lock:
            return self.contracts.get(key)

    def trigger_death(self, key: str) -> bool:
        """Manually trigger death of a cache entry"""
        with self.lock:
            contract = self.contracts.get(key)
            if contract and contract.death_callback:
                try:
                    contract.death_callback()
                    return True
                except Exception as e:
                    print(f"⚠️ Error in death callback: {e}")
            return False

# ZEPHYRUS: Authority - Command Hierarchy
#=============================================================
=

@dataclass
class AuthorityChain:
    """Who can override you?"""
    owner_id: str
    creator_id: str
    authority_level: int  # Higher = more authority (0-100)
    required_level_read: int = 0
    required_level_write: int = 50
    required_level_delete: int = 75
    root_override_allowed: bool = True
```

```
class Zephyrus:
    """
    OPERATOR: BRIDGE
    Authority hierarchy system - "Who can override you?"
    Root access and command control
    """

    def __init__(self, root_authority_level: int = 100):
        self.user_authorities: Dict[str, int] = {}
        self.key_authorities: Dict[str, AuthorityChain] = {}
        self.root_level = root_authority_level
        self.lock = threading.Lock()

    def set_user_authority(self, user_id: str, level: int):
        """Set authority level for a user (0-100)"""
        with self.lock:
            self.user_authorities[user_id] = max(0, min(100, level))

    def bind_authority(self, key: str, chain: AuthorityChain):
        """Bind authority chain to a key"""
        with self.lock:
            self.key_authorities[key] = chain

    def can_read(self, user_id: str, key: str) -> bool:
        """Check if user can read this key"""
        return self._check_permission(user_id, key, "read")

    def can_write(self, user_id: str, key: str) -> bool:
        """Check if user can write this key"""
        return self._check_permission(user_id, key, "write")

    def can_delete(self, user_id: str, key: str) -> bool:
        """Check if user can delete this key"""
        return self._check_permission(user_id, key, "delete")

    def _check_permission(self, user_id: str, key: str, action: str) -> bool:
        """Internal permission check"""
        with self.lock:
            user_level = self.user_authorities.get(user_id, 0)
            chain = self.key_authorities.get(key)

            if not chain:
                return True  # No authority chain = public access
            # Owner always has access
            if user_id == chain.owner_id:
                return True

            # Root override check
            if chain.root_override_allowed and user_level >= self.root_level:
                return True

            # Check required level
            if action == "read":
                return user_level >= chain.required_level_read
            elif action == "write":
                return user_level >= chain.required_level_write
            elif action == "delete":
                return user_level >= chain.required_level_delete

            return False

    def get_authority(self, key: str) -> Optional[AuthorityChain]:
        """Get authority chain for a key"""
        with self.lock:
            return self.key_authorities.get(key)

# HERAIA: Order/Structure - Role-Based Access Control
#============================================================================
=

@dataclass
class Role:
    """A role in the system"""
    name: str
    authority_level: int
    permissions: Set[str] = field(default_factory=set)
    description: str = ""

class Heraia:
    """
    OPERATOR: WRAP
    Role-based access control and governance
    """

    def __init__(self):
        self.roles: Dict[str, Role] = {}
        self.user_roles: Dict[str, Set[str]] = defaultdict(set)
        self.lock = threading.Lock()

        # Initialize default roles
        self._initialize_default_roles()

    def _initialize_default_roles(self):
        """Initialize default role hierarchy"""
        self.define_role(Role(
            name="admin",
            authority_level=100,
            permissions={"read", "write", "delete", "govern"},
            description="Full system access"
        ))

        self.define_role(Role(
            name="developer",
            authority_level=75,
            permissions={"read", "write", "delete", "govern"},
            description="Can modify cache entries and audit"
        ))

        self.define_role(Role(
            name="operator",
            authority_level=50,
            permissions={"read", "write"},
            description="Can read and write cache"
        ))

        self.define_role(Role(
            name="viewer",
            authority_level=10,
            permissions={"read"},
            description="Read-only access"
        ))

    def define_role(self, role: Role):
        """Define a new role"""
        with self.lock:
            self.roles[role.name] = role

    def assign_role(self, user_id: str, role_name: str):
        """Assign a role to a user"""
        with self.lock:
            if role_name in self.roles:
                self.user_roles[user_id].add(role_name)

    def revoke_role(self, user_id: str, role_name: str):
        """Revoke a role from a user"""
        with self.lock:
            if user_id in self.user_roles:
                self.user_roles[user_id].discard(role_name)

    def has_permission(self, user_id: str, permission: str) -> bool:
        """Check if user has a specific permission through any role"""
        with self.lock:
            user_role_names = self.user_roles.get(user_id, set())
            for role_name in user_role_names:
                role = self.roles.get(role_name)
                if role and permission in role.permissions:
                    return True
            return False

    def get_max_authority(self, user_id: str) -> int:
        """Get the maximum authority level from all user's roles"""
        with self.lock:
            user_role_names = self.user_roles.get(user_id, set())
            max_auth = 0
            for role_name in user_role_names:
                role = self.roles.get(role_name)
                if role:
                    max_auth = max(max_auth, role.authority_level)
            return max_auth

    def get_user_roles(self, user_id: str) -> Set[str]:
        """Get all roles for a user"""
        with self.lock:
            return set(self.user_roles.get(user_id, set()))

#============================================================================
=
# GOVERNED CACHE ENTRY
#============================================================================
=

@dataclass
class GovernedCacheEntry:
    """
    OPERATOR: EMERGE
```

```
Complete cache entry with full governance metadata
Answers all the key questions:
1. What spell created you? → genesis (LOGORA)
2. What cloth modified you? → provenance (REVELA)
3. What law manual approved you? → policy (MA'ATARA)
4. What would make you obsolete? → lifecycle (MOIRAE)
5. Who can override you? → authority (ZEPHYRUS + HERAIA)
"""
# Core data
key: str
value: Any

# Phase 1: Cache invalidation metadata
version: int
timestamp: float
dependencies: Set[str] = field(default_factory=set)

# Phase 2: Governance metadata
genesis: Optional[CreationRecord] = None           # LOGORA
provenance: List[ModificationRecord] = field(default_factory=list)  # REVELA
policy: Optional[PolicyBinding] = None             # MA'ATARA
lifecycle: Optional[LifecycleContract] = None      # MOIRAE
authority: Optional[AuthorityChain] = None         # ZEPHYRUS

def is_compliant(self) -> bool:
    """Check if entry complies with all policies"""
    if not self.policy:
        return True
    return self.policy.compliance_status == "COMPLIANT"

def is_expired(self, moirae: 'Moirae') -> bool:
    """Check if lifecycle has expired"""
    if not self.lifecycle:
        return False
    expired, _ = moirae.is_expired(self.key)
    return expired

def can_be_accessed_by(self, user_id: str, action: str,
                      zephyrus: 'Zephyrus', heraia: 'Heraia') -> bool:
    """Check if user has permission for action"""
    # Check role-based permissions first
    if not heraia.has_permission(user_id, action):
        return False
    # Check authority chain
    if action == "read":
        return zephyrus.can_read(user_id, self.key)
    elif action == "write":
        return zephyrus.can_write(user_id, self.key)
    elif action == "delete":
        return zephyrus.can_delete(user_id, self.key)

    return False

def audit_trail(self) -> str:
    """Generate complete audit trail"""
    lines = []
    lines.append(f"=== AUDIT TRAIL: {self.key} ===")

    if self.genesis:
        lines.append(f"\n📜 GENESIS (LOGORA):")
        lines.append(f"   {self.genesis}")

    if self.provenance:
        lines.append(f"\n🔄 PROVENANCE (REVELA):")
        for i, mod in enumerate(self.provenance[-5:]):  # Last 5 modifications
            lines.append(f"   {i+1}. {mod}")

    if self.policy:
        lines.append(f"\n⚖️  POLICY (MA'ATARA):")
        lines.append(f"   Status: {self.policy.compliance_status}")
        lines.append(f"   Rules: {len(self.policy.rules)}")
        if self.policy.violations:
            lines.append(f"   Violations: {', '.join(self.policy.violations)}")

    if self.lifecycle:
        lines.append(f"\n⏳ LIFECYCLE (MOIRAE):")
        age = time.time() - self.lifecycle.birth_timestamp
        lines.append(f"   Age: {age:.2f}s")
        if self.lifecycle.max_age_seconds:
            lines.append(f"   Max age: {self.lifecycle.max_age_seconds}s")

    if self.authority:
        lines.append(f"\n👑 AUTHORITY (ZEPHYRUS):")
        lines.append(f"   Owner: {self.authority.owner_id}")
        lines.append(f"   Creator: {self.authority.creator_id}")
        lines.append(f"   Authority level: {self.authority.authority_level}")
    return "\n".join(lines)

#==============================================================================
=
# ORACLE_CACHE V2: Production-Grade Governed Cache
#==============================================================================
=

class OracleCacheV2:
"""
OPERATOR: FINALIZE

EMERGE(
    # Phase 1: Core cache invalidation
    Chronom + Relata + Echo + Hydrina +
    Byzantium + Entangla + Oedipha + Deadpoolia +

    # Phase 2: Governance layer
    Logora + Revela + Ma'atara + Moirae + Zephyrus + Heraia
)

Emergent Properties:
    Phase 1:
    - Temporal Consensus
    - Predictive Coherence
    - Self-Healing Dependencies
    - Quantum Invalidation
    - Regenerative Cascade

    Phase 2:
    - Creation Attribution
    - Modification Provenance
    - Policy Enforcement
    - Lifecycle Management
    - Authority Control
    - Role-Based Access
"""

def __init__(self, node_id: str = "node_0", current_user: str = "system"):
    # Phase 1: Core components
    self.chronom = Chronom()
    self.relata = Relata()
    self.echo = Echo()
    self.hydrina = Hydrina()
    self.byzantium = Byzantium(node_id)
    self.entangla = Entangla()
    self.oedipha = Oedipha()
    self.deadpoolia = Deadpoolia(self.relata)

    # Phase 2: Governance components
    self.logora = Logora()
    self.revela = Revela()
    self.ma_atara = Ma_atara()
    self.moirae = Moirae()
    self.zephyrus = Zephyrus()
    self.heraia = Heraia()

    # Storage
    self.cache: Dict[str, GovernedCacheEntry] = {}
    self.lock = threading.Lock()
    self.current_user = current_user

    # Initialize system user with admin role
    self.heraia.assign_role("system", "admin")
    self.zephyrus.set_user_authority("system", 100)

    # Subscribe to invalidation events
    self.echo.subscribe(self._handle_invalidation)

def set_current_user(self, user_id: str):
    """Set the current user context"""
    self.current_user = user_id

def set(self, key: str, value: Any,
        depends_on: Set[str] = None,
        regenerator: Callable[[], Any] = None,
        lifecycle: LifecycleContract = None,
        policies: List[PolicyRule] = None,
        authority: AuthorityChain = None) -> bool:
    """
    OPERATOR: EMERGE
    Set cache entry with full governance
    """
    # Check if current user has write permission
    if key in self.cache:
        if not self.zephyrus.can_write(self.current_user, key):
            print(f"❌ Permission denied: {self.current_user} cannot write to {key}")
    return False

# Validate against policies
if policies:
for policy in policies:
    self.ma_atara.add_key_policy(key, policy)

is_valid, violations = self.ma_atara.validate(key, value)
if not is_valid:
print(f"❌ Policy violation for {key}: {violations}")
return False

# Get old value for provenance tracking
old_value = None
if key in self.cache:
old_value = self.cache[key].value

with self.lock:
# Create governed cache entry
entry = GovernedCacheEntry(
    key=key,
    value=value,
    version=0,
    timestamp=time.time(),
    dependencies=depends_on or set()
)

# LOGORA: Record creation
if key not in self.cache:  # New entry
    self.logora.record_creation(
        key=key,
        spell="EMERGE(Chronom + Hydrina + Governance)",
        operators=["EMERGE", "LAYER", "WRAP"],
        creator_id=self.current_user,
        context={"dependencies": list(depends_on or set())}
    )
    entry.genesis = self.logora.get_genesis(key)
else:
    # Keep existing genesis
    entry.genesis = self.cache[key].genesis

# REVELA: Record modification
if old_value is not None:
    self.revela.record_modification(
        key=key,
        spell="set",
        operator="LAYER",
        modifier_id=self.current_user,
        reason="manual_update",
        old_value=old_value,
        new_value=value
    )
entry.provenance = self.revela.get_provenance(key)

# MA'ATARA: Bind policy
entry.policy = PolicyBinding(
    rules=list(self.ma_atara.key_policies.get(key, [])),
    compliance_status="COMPLIANT" if is_valid else "VIOLATED",
    last_audit=time.time(),
    violations=violations
)

# MOIRAE: Bind lifecycle
if lifecycle:
    self.moirae.bind_lifecycle(key, lifecycle)
    entry.lifecycle = lifecycle

# ZEPHYRUS: Bind authority
if authority:
    self.zephyrus.bind_authority(key, authority)
    entry.authority = authority
elif key not in self.cache:  # New entry without explicit authority
    # Create default authority chain
    default_authority = AuthorityChain(
        owner_id=self.current_user,
        creator_id=self.current_user,
        authority_level=self.heraia.get_max_authority(self.current_user)
    )
    self.zephyrus.bind_authority(key, default_authority)
    entry.authority = default_authority

# Store entry
self.cache[key] = entry

# Phase 1 tracking
version = self.chronom.snapshot(key, value, depends_on or set())
entry.version = version
```

```
        if depends_on:
            for dep in depends_on:
                self.relata.add_dependency(key, dep)

        if regenerator:
            self.hydrina.register_regenerator(key, regenerator)

        self.oedipha.record_update(key)
        self.byzantium.vote(key, ConsensusVote.VALID)

    return True

def get(self, key: str, as_user: str = None) -> Optional[Any]:
    """
    OPERATOR: BRIDGE
    Get cache entry with full governance checks
    """
    user = as_user or self.current_user

    # Check read permission
    if not self.zephyrus.can_read(user, key):
        print(f"❌ Permission denied: {user} cannot read {key}")
        return None

    # Check if expired
    if key in self.cache:
        expired, reason = self.moirae.is_expired(key)
        if expired:
            print(f"⏳ Entry expired: {key} ({reason})")
            self.invalidate(key, reason=f"lifecycle_expired: {reason}")
            return None

    # Check predictive staleness
    predicted_stale = self.oedipha.predict_staleness_time(key)
    if predicted_stale and time.time() >= predicted_stale:
        self._invalidate_key_simple(key, "predictive_staleness")
        return None

    # Check consensus
    consensus = self.byzantium.get_consensus(key)
    if consensus == ConsensusVote.STALE:
        return None

    # Record access
```

```
        self.oedipha.record_access(key)

        with self.lock:
            entry = self.cache.get(key)
            return entry.value if entry else None

    def get_entry(self, key: str, as_user: str = None) -> Optional[GovernedCacheEntry]:
        """Get full governed cache entry (requires elevated permissions)"""
        user = as_user or self.current_user

        # Requires 'govern' permission to see full metadata
        if not self.heraia.has_permission(user, "govern"):
            print(f"❌ Permission denied: {user} cannot access governance metadata")
            return None

        with self.lock:
            return self.cache.get(key)

    def invalidate(self, key: str, reason: str = "manual"):
        """
        OPERATOR: EMERGE
        Invalidate with governance checks and full cascade
        """
        # Check delete permission
        if not self.zephyrus.can_delete(self.current_user, key):
            print(f"❌ Permission denied: {self.current_user} cannot invalidate {key}")
            return

        # Get cascade targets
        cascade_targets = self.relata.get_cascade_targets(key)
        entangled = self.entangla.get_entangled(key)
        all_targets = list(set(cascade_targets + list(entangled) + [key]))

        print(f"\n🔥 INVALIDATION CASCADE for '{key}'")
        print(f"   User: {self.current_user}")
        print(f"   Reason: {reason}")
        print(f"   Targets: {all_targets}")

        # Assess damage and compute phases
        phases = self.deadpoolia.assess_damage(set(all_targets))

        if not phases:
            print("   ⚠️ Cannot regenerate: circular dependency detected")
            return
        print(f"   📊 Regeneration phases: {len(phases)}")

        # Execute regeneration in phase order
        for phase in phases:
            print(f"\n   Phase {phase.phase_number}: {phase.keys}")

            for target_key in phase.keys:
                if self.deadpoolia.check_dependencies(target_key, set(all_targets)):
                    self._regenerate_key(target_key, reason)
                    self.deadpoolia.mark_ready(target_key)
                    print(f"      ✓ Regenerated: {target_key}")
                else:
                    print(f"      ✗ Blocked: {target_key}")

        # Broadcast completion
        event = InvalidationEvent(
            key=key,
            timestamp=time.time(),
            cascade_targets=all_targets,
            reason=reason
        )
        self.echo.broadcast(event)

        print(f"\n✅ REGENERATIVE CASCADE COMPLETE\n")

    def _handle_invalidation(self, event: InvalidationEvent):
        """Handle invalidation event from broadcast"""
        pass

    def _invalidate_key_simple(self, key: str, reason: str):
        """Simple invalidation without cascade"""
        with self.lock:
            if key in self.cache:
                del self.cache[key]
            self.byzantium.vote(key, ConsensusVote.STALE)

    def _regenerate_key(self, key: str, reason: str):
        """
        OPERATOR: CHAIN
        Regenerate key with governance tracking
        """
        old_value = None
        if key in self.cache:
```

```
    old_value = self.cache[key].value

new_value = self.hydrina.regenerate(key)

if new_value is not None:
    # Record regeneration as modification
    self.revela.record_modification(
        key=key,
        spell="Deadpoolia",
        operator="CHAIN",
        modifier_id="system",
        reason=f"auto_regeneration: {reason}",
        old_value=old_value,
        new_value=new_value
    )

    with self.lock:
        if key in self.cache:
            # Update existing entry
            entry = self.cache[key]
            entry.value = new_value
            entry.timestamp = time.time()
            entry.provenance = self.revela.get_provenance(key)

            # CRITICAL: Refresh lifecycle and authority from governance layer
            entry.lifecycle = self.moirae.get_contract(key)
            entry.authority = self.zephyrus.get_authority(key)

            version = self.chronom.current_version.get(key, 0)
            deps = self.chronom.get_dependencies(key, version)
            self.chronom.snapshot(key, new_value, deps)

            self.byzantium.vote(key, ConsensusVote.VALID)
            self.oedipha.record_update(key)
        else:
            # CREATE new entry if it doesn't exist
            version = self.chronom.current_version.get(key, 0)
            deps = self.chronom.get_dependencies(key, version)

            entry = GovernedCacheEntry(
                key=key,
                value=new_value,
                version=version + 1,
                timestamp=time.time(),
                dependencies=deps
            )

            # Restore governance metadata
            entry.genesis = self.logora.get_genesis(key)
            entry.provenance = self.revela.get_provenance(key)
            entry.lifecycle = self.moirae.get_contract(key)
            entry.authority = self.zephyrus.get_authority(key)

            # Get policies from Ma'atara
            entry.policy = PolicyBinding(
                rules=list(self.ma_atara.key_policies.get(key, [])),
                compliance_status="COMPLIANT",
                last_audit=time.time(),
                violations=[]
            )

            self.cache[key] = entry
            self.chronom.snapshot(key, new_value, deps)
            self.byzantium.vote(key, ConsensusVote.VALID)
            self.oedipha.record_update(key)

    def audit(self, key: str) -> Optional[str]:
        """
        Generate complete audit trail for a cache entry
        Requires 'govern' permission
        """
        if not self.heraia.has_permission(self.current_user, "govern"):
            print(f"❌ Permission denied: {self.current_user} cannot audit")
            return None

        entry = self.get_entry(key)
        if entry:
            return entry.audit_trail()
        return None

    def entangle_keys(self, key1: str, key2: str):
        """Create quantum entanglement between keys"""
        self.entangla.entangle(key1, key2)

#============================================================================
=
# FINALIZE: Demonstration
```

```
#==========================================================================
=

def demonstrate_governed_cache():
    """
    OPERATOR: FINALIZE
    Demonstrate production-grade governed cache
    """
    print("=" * 80)
    print("ORACLE_CACHE V2: Production-Grade Governed Distributed Cache")
    print("=" * 80)

    # Create cache
    cache = OracleCacheV2(node_id="prod_node", current_user="system")

    # Create users with different roles
    print("\n👥 Setting up users and roles...")
    cache.heraia.assign_role("alice", "developer")
    cache.heraia.assign_role("bob", "operator")
    cache.heraia.assign_role("charlie", "viewer")

    cache.zephyrus.set_user_authority("alice", 75)
    cache.zephyrus.set_user_authority("bob", 50)
    cache.zephyrus.set_user_authority("charlie", 10)

    print("   ✓ alice   → developer (authority: 75)")
    print("   ✓ bob     → operator  (authority: 50)")
    print("   ✓ charlie → viewer    (authority: 10)")

    # Define policies
    print("\n⚖️  Defining policies...")

    def not_empty(value):
        return value is not None and value != {}

    def has_timestamp(value):
        return isinstance(value, dict) and 'updated' in value

    age_policy = PolicyRule(
        name="must_have_timestamp",
        condition=has_timestamp,
        enforcement_level="STRICT",
        violation_action="REJECT",
        description="All user data must have timestamp"
```

```
)

cache.ma_atara.add_global_policy(PolicyRule(
    name="not_empty",
    condition=not_empty,
    enforcement_level="STRICT",
    violation_action="REJECT"
))

# Define regenerators
def gen_user_data():
    return {"name": "Alice", "age": 30, "updated": time.time()}

def gen_user_profile():
    return {"bio": "Software Engineer", "updated": time.time()}

def gen_user_settings():
    return {"theme": "dark", "notifications": True, "updated": time.time()}

# Set cache entries with governance
print("\n📝 Creating governed cache entries...")
cache.set_current_user("alice")

# User data with lifecycle (expires in 10 seconds)
cache.set(
    "user:1:data",
    gen_user_data(),
    regenerator=gen_user_data,
    lifecycle=LifecycleContract(max_age_seconds=10.0, auto_cleanup=True),
    policies=[age_policy],
    authority=AuthorityChain(
        owner_id="alice",
        creator_id="alice",
        authority_level=75,
        required_level_write=50,
        required_level_delete=75
    )
)

cache.set(
    "user:1:profile",
    gen_user_profile(),
    depends_on={"user:1:data"},
    regenerator=gen_user_profile,
    policies=[age_policy]
)

cache.set(
    "user:1:settings",
    gen_user_settings(),
    depends_on={"user:1:data"},
    regenerator=gen_user_settings,
    policies=[age_policy]
)

print("   ✓ user:1:data     (owner: alice, lifecycle: 10s)")
print("   ✓ user:1:profile  (depends on: user:1:data)")
print("   ✓ user:1:settings (depends on: user:1:data)")

# Entangle profile and settings
print("\n🔗 Entangling profile and settings...")
cache.entangle_keys("user:1:profile", "user:1:settings")

# Test access control
print("\n" + "=" * 80)
print("TESTING ACCESS CONTROL")
print("=" * 80)

print("\n1️⃣ Charlie (viewer) tries to read user:1:data:")
result = cache.get("user:1:data", as_user="charlie")
print(f"   Result: {result}")

print("\n2️⃣ Bob (operator) tries to write user:1:data:")
cache.set_current_user("bob")
success = cache.set("user:1:data", {"name": "Bob", "age": 25, "updated": time.time()})
print(f"   Success: {success}")

print("\n3️⃣ Charlie (viewer) tries to delete user:1:data:")
cache.set_current_user("charlie")
cache.invalidate("user:1:data", reason="charlie_attempted_delete")

print("\n4️⃣ Alice (owner) reads all entries:")
cache.set_current_user("alice")
data = cache.get("user:1:data")
profile = cache.get("user:1:profile")
settings = cache.get("user:1:settings")
print(f"   Data:     {data}")
print(f"   Profile:  {profile}")
print(f"   Settings: {settings}")

# Test audit trail
print("\n" + "=" * 80)
print("AUDIT TRAIL")
print("=" * 80)

cache.set_current_user("alice")  # Only developers/admins can audit
audit = cache.audit("user:1:data")
if audit:
    print(audit)

# Test invalidation with governance
print("\n" + "=" * 80)
print("TESTING GOVERNED INVALIDATION CASCADE")
print("=" * 80)

time.sleep(0.1)

cache.set_current_user("alice")
cache.invalidate("user:1:data", reason="alice_manual_update")

# Verify regeneration
print("\n📊 After governed cascade:")
data = cache.get("user:1:data")
profile = cache.get("user:1:profile")
settings = cache.get("user:1:settings")
print(f"   Data:     {data}")
print(f"   Profile:  {profile}")
print(f"   Settings: {settings}")

# Final summary
print("\n" + "=" * 80)
print("EMERGENT PROPERTIES DEMONSTRATED")
print("=" * 80)
print("\nPhase 1 - Cache Invalidation:")
print("  ✓ Temporal Consensus      (Chronom + Byzantium)")
print("  ✓ Predictive Coherence    (Oedipha)")
print("  ✓ Dependency Tracking     (Relata)")
print("  ✓ Regenerative Cascade    (Deadpoolia)")
print("  ✓ Auto-Healing            (Hydrina)")
print("  ✓ Quantum Invalidation    (Entangla + Echo)")

print("\nPhase 2 - Governance Layer:")
```

```
    print("  ✓ Creation Attribution    (Logora)")
    print("  ✓ Modification Provenance (Revela)")
    print("  ✓ Policy Enforcement      (Ma'atara)")
    print("  ✓ Lifecycle Management    (Moirae)")
    print("  ✓ Authority Control       (Zephyrus)")
    print("  ✓ Role-Based Access       (Heraia)")

    print("\n" + "=" * 80)
    print("🎯 Production-Grade Governed Cache - COMPLETE")
    print("=" * 80)

    # Answer the key questions
    print("\n" + "=" * 80)
    print("ANSWERING THE KEY QUESTIONS")
    print("=" * 80)

    entry = cache.get_entry("user:1:data")
    if entry:
        print(f"\n❓ What spell created you?")
        print(f"   → {entry.genesis}")

        print(f"\n❓ What cloth modified you?")
        if entry.provenance:
            print(f"   → Last modification: {entry.provenance[-1]}")

        print(f"\n❓ What law manual approved you?")
        print(f"   → Policy status: {entry.policy.compliance_status}")
        print(f"   → Rules: {len(entry.policy.rules)}")

        print(f"\n❓ What would make you obsolete?")
        if entry.lifecycle:
            print(f"   → Max age: {entry.lifecycle.max_age_seconds}s")
            print(f"   → Current age: {time.time() - entry.lifecycle.birth_timestamp:.2f}s")

        print(f"\n❓ Who can override you?")
        if entry.authority:
            print(f"   → Owner: {entry.authority.owner_id}")
            print(f"   → Required write level: {entry.authority.required_level_write}")
            print(f"   → Required delete level: {entry.authority.required_level_delete}")

    print("\n" + "=" * 80)

if __name__ == "__main__":
    demonstrate_governed_cache()
```

```
ORACLE_CACHE V2: Production-Grade Governed Distributed Cache
======================================================================
========

👥 Setting up users and roles...
  ✓ alice   → developer (authority: 75)
  ✓ bob     → operator  (authority: 50)
  ✓ charlie → viewer    (authority: 10)

⚖️  Defining policies...

📝 Creating governed cache entries...
  ✓ user:1:data     (owner: alice, lifecycle: 10s)
  ✓ user:1:profile  (depends on: user:1:data)
  ✓ user:1:settings (depends on: user:1:data)

🔗 Entangling profile and settings...

======================================================================
========
TESTING ACCESS CONTROL
======================================================================
========

1️⃣ Charlie (viewer) tries to read user:1:data:
   Result: {'name': 'Alice', 'age': 30, 'updated': 1769469938.2097173}

2️⃣ Bob (operator) tries to write user:1:data:
   Success: True

3️⃣ Charlie (viewer) tries to delete user:1:data:
❌ Permission denied: charlie cannot invalidate user:1:data

4️⃣ Alice (owner) reads all entries:
   Data:     {'name': 'Bob', 'age': 25, 'updated': 1769469938.2107265}
   Profile:  {'bio': 'Software Engineer', 'updated': 1769469938.2097173}
   Settings: {'theme': 'dark', 'notifications': True, 'updated': 1769469938.2097173}

======================================================================
========
AUDIT TRAIL
======================================================================
========
=== AUDIT TRAIL: user:1:data ===

📜 GENESIS (LOGORA):
  Created by alice using EMERGE(Chronom + Hydrina + Governance) at 2026-01-26
23:25:38.209717

🔄 PROVENANCE (REVELA):
  1. bob modified via set (manual_update) at 2026-01-26 23:25:38.210726

⚖️ POLICY (MA'ATARA):
  Status: COMPLIANT
  Rules: 1

======================================================================
========
TESTING GOVERNED INVALIDATION CASCADE
======================================================================
========

🔥 INVALIDATION CASCADE for 'user:1:data'
  User: alice
  Reason: alice_manual_update
  Targets: ['user:1:settings', 'user:1:data', 'user:1:profile']
  📊 Regeneration phases: 2

  Phase 0: ['user:1:data']
    ✓ Regenerated: user:1:data

  Phase 1: ['user:1:settings', 'user:1:profile']
    ✓ Regenerated: user:1:settings
    ✓ Regenerated: user:1:profile

✅ REGENERATIVE CASCADE COMPLETE


📊 After governed cascade:
  Data:     {'name': 'Alice', 'age': 30, 'updated': 1769469938.3630629}
  Profile:  {'bio': 'Software Engineer', 'updated': 1769469938.364215}
  Settings: {'theme': 'dark', 'notifications': True, 'updated': 1769469938.364215}

======================================================================
========
EMERGENT PROPERTIES DEMONSTRATED
```

```
================================================================================
========

Phase 1 - Cache Invalidation:
  ✓ Temporal Consensus      (Chronom + Byzantium)
  ✓ Predictive Coherence    (Oedipha)
  ✓ Dependency Tracking     (Relata)
  ✓ Regenerative Cascade    (Deadpoolia)
  ✓ Auto-Healing            (Hydrina)
  ✓ Quantum Invalidation    (Entangla + Echo)

Phase 2 - Governance Layer:
  ✓ Creation Attribution    (Logora)
  ✓ Modification Provenance (Revela)
  ✓ Policy Enforcement      (Ma'atara)
  ✓ Lifecycle Management    (Moirae)
  ✓ Authority Control       (Zephyrus)
  ✓ Role-Based Access       (Heraia)

================================================================================
========
🎯 Production-Grade Governed Cache - COMPLETE
================================================================================
========

================================================================================
========
ANSWERING THE KEY QUESTIONS
================================================================================
========

❓ What spell created you?
   → Created by alice using EMERGE(Chronom + Hydrina + Governance) at 2026-01-26
23:25:38.209717

❓ What cloth modified you?
   → Last modification: system modified via Deadpoolia (auto_regeneration:
alice_manual_update) at 2026-01-26 23:25:38.363063

❓ What law manual approved you?
   → Policy status: COMPLIANT
   → Rules: 1

❓ What would make you obsolete?
```

```
  → Max age: 10.0s
  → Current age: 0.26s

❓ Who can override you?
  → Owner: alice
  → Required write level: 50
  → Required delete level: 75

======================================================================
========
```