Integrates persistence, context management, and ingestion scaffolds—providing a complete starting point for the PlayNAC "KERNEL".

---

src/kernel/config.py

```python
import os
from typing import Any, List

class ConfigManager:
    """
    Loads environment files, validates required keys, and provides getters.

    Supports multiple .env files for different environments.
    """
    def __init__(self, env_files: List[str] = [".env"]):
        self.env_files = env_files
        self._loaded = False

    def load_env(self) -> None:
        if self._loaded:
            return
        for file in self.env_files:
```

```
            if os.path.isfile(file):
                with open(file) as f:
                    for line in f:
                        if line.strip().startswith('#') or '=' not in line:
                            continue
                        key, val = line.strip().split('=', 1)
                        os.environ.setdefault(key, val)
        self._loaded = True

    def validate(self, required_keys: List[str]) -> None:
        missing = [k for k in required_keys if k not in os.environ]
        if missing:
            raise KeyError(f"Missing required config keys: {missing}")

    def get(self, key: str, default: Any = None) -> Any:
        return os.environ.get(key, default)
```

---

src/kernel/models.py

```python
from dataclasses import dataclass
from typing import Any
```

```
@dataclass
class Block:
    index: int
    timestamp: float
    data: Any
    previous_hash: str
    nonce: int
    hash: str
```

---

src/kernel/storage.py

```python
"""
Persistent storage for blockchain and tasks using SQLite.
"""
import sqlite3
import json

class Storage:
    """
    Persists and loads Block records in SQLite.
```

```
    """
    def __init__(self, path: str = 'playnac.db'):
        self.conn = sqlite3.connect(path)
        self._init_schema()

    def _init_schema(self):
        c = self.conn.cursor()
        c.execute('''
            CREATE TABLE IF NOT EXISTS blocks (
                idx INTEGER PRIMARY KEY,
                timestamp REAL,
                data TEXT,
                prev_hash TEXT,
                nonce INTEGER,
                hash TEXT
            )
        ''')
        self.conn.commit()

    def save_block(self, block):
        c = self.conn.cursor()
        c.execute(
            'INSERT INTO blocks(idx,timestamp,data,prev_hash,nonce,hash) VALUES
(?,?,?,?,?,?)',
            (
```

```
                block.index,
                block.timestamp,
                json.dumps(block.data),
                block.previous_hash,
                block.nonce,
                block.hash,
            )
        )
        self.conn.commit()

    def load_blocks(self):
        """
        Load all persisted blocks ordered by index.
        Returns list of dicts matching Block fields.
        """
        c = self.conn.cursor()
        c.execute('SELECT idx,timestamp,data,prev_hash,nonce,hash FROM blocks ORDER BY
idx')
        rows = c.fetchall()
        blocks = []
        for idx, ts, data_json, prev_hash, nonce, h in rows:
            blocks.append({
                'index': idx,
                'timestamp': ts,
                'data': json.loads(data_json),
```

```
            'previous_hash': prev_hash,
            'nonce': nonce,
            'hash': h,
        })
    return blocks
```

---

src/kernel/context_manager.py

```python
"""
Session and context manager for PlayNAC KERNEL.
"""
from typing import Dict, Any

class ContextManager:
    """
    Maintains per-session state across modules for multi-turn Q&A and simulations.
    """
    def __init__(self):
        self.sessions: Dict[str, Dict[str, Any]] = {}

    def get_context(self, session_id: str) -> Dict[str, Any]:
```

```
        return self.sessions.setdefault(session_id, {})

    def update_context(self, session_id: str, key: str, value: Any) -> None:
        ctx = self.get_context(session_id)
        ctx[key] = value
```

---

src/kernel/playnac_kernel.py

```python
import time
import hashlib
from typing import Optional, List, Any
from .config import ConfigManager
from .models import Block
from .storage import Storage
from bee.bio_pow import BioPoW
from earnedpath.simulation_engine import SimulationEngine
from berc.jas_consensus import JASConsensus
from media.media_processor import MediaProcessor
from berc.models import MediaTask

class PlayNACKernel:
```

```
"""
Core orchestrator for ERES PlayNAC "KERNEL" with persistence.

Bootstraps modules, loads persisted blocks, manages mining, and saves new blocks.
"""
def __init__(self):
    # Load config
    self.config = ConfigManager(env_files=[".env", ".env.local"])
    self.config.load_env()
    self.config.validate(["WEB3_RPC_URL", "BEE_SECRET_KEY", "DB_PATH"])

    # Storage
    db_path = self.config.get("DB_PATH")
    self.storage = Storage(path=db_path)

    # Engines
    self.bio_pow = BioPoW(secret_key=self.config.get("BEE_SECRET_KEY"))
    self.sim_engine = SimulationEngine()
    self.media_processor = MediaProcessor()
    self.consensus = JASConsensus()

    # Load chain
    self.blockchain: List[Block] = []
    for rec in self.storage.load_blocks():
```

```
            self.blockchain.append(Block(**rec))
        # TODO: rebuild consensus graph

        self.task_queue: List[MediaTask] = []

    def submit_media_task(self, task: MediaTask) -> None:
        """Queue a media processing task for mining."""
        self.task_queue.append(task)

    def _calculate_hash(self, index: int, ts: float, data: Any,
                        prev_hash: str, nonce: int) -> str:
        s = f"{index}{ts}{data}{prev_hash}{nonce}".encode()
        return hashlib.sha256(s).hexdigest()

    def _get_difficulty_target(self) -> float:
        last = [b.data['ep'] for b in self.blockchain[-5:]]
        return sum(last)/len(last) if last else 0.5

    def mine_block(self, max_nonce: int = 10000) -> Optional[Block]:
        """
        Mine a block: BioPoW -> media -> block -> persist -> consensus link.
        """
        if not self.task_queue:
            return None
```

```
task = self.task_queue.pop(0)
task.ep_value = self.bio_pow.generate_ep()

for nonce in range(max_nonce):
    task.nonce = nonce
    try:
        frame = self.media_processor.process_media_task(task)
    except ValueError:
        continue

    target = self._get_difficulty_target()
    if not self.bio_pow.validate(task.ep_value, target):
        continue

    idx = len(self.blockchain)
    ts = time.time()
    data = {
        'task_id': task.id,
        'ep': task.ep_value,
        'media_hash': hashlib.sha256(frame.tobytes()).hexdigest()
    }
    prev = self.blockchain[-1].hash if self.blockchain else '0'*64
    h = self._calculate_hash(idx, ts, data, prev, nonce)
    block = Block(idx, ts, data, prev, nonce, h)
```

```
            self.blockchain.append(block)

            # persist
            self.storage.save_block(block)

            # consensus
            if idx>0:
                prev_id = self.blockchain[idx-1].data['task_id']
                self.consensus.create_link(MediaTask(prev_id,None,'',0,0), task, task.ep_value)
            return block
        return None

    def run(self, iterations: int = 1) -> None:
        for _ in range(iterations):
            b = self.mine_block()
            if b:
                print(f"Mined {b.index} EP={b.data['ep']:.3f} hash={b.hash[:8]}")

if __name__=='__main__':
    from berc.models import MediaTask
    k = PlayNACKernel()
    k.submit_media_task(MediaTask('t0', b'd', 'type', 0, time.time()))
    k.run(1)
```

---

tests/test_storage.py (updated)

```python
import sqlite3
import pytest
from src.kernel.storage import Storage


 def test_load_and_save_roundtrip(tmp_path):
    from src.kernel.models import Block
    path = tmp_path / 'db.db'
    s = Storage(path=str(path))
    blk = Block(5, 1.23, {'x':1}, '0'*64, 7, 'h')
    s.save_block(blk)
    rows = s.load_blocks()
    assert len(rows)==1
    assert rows[0]['index']==5
    assert rows[0]['data']=={'x':1}
```

With v6.2 in place, you can now persist and reload the chain seamlessly, and tests verify correct storage behavior. Next up: rebuilding consensus graph on load or integrating deeper domain logic.