

```
# File: src/kernel/config.py

"""

Configuration management for PlayNAC Kernel

"""

import os

from typing import Any, Dict, List


class ConfigManager:

    """

    Loads .env, validates required keys, and provides getters.

    """

    def __init__(self, env_file: str = ".env"):

        self.env_file = env_file

        self.config: Dict[str, Any] = {}


    def load_env(self) -> None:

        # Load key/value pairs from .env into os.environ

        with open(self.env_file) as f:

            for line in f:

                if '=' in line:

                    key, val = line.strip().split('=', 1)

                    os.environ.setdefault(key, val)


    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)
```

```
# File: src/kernel/playnac_kernel.py
```

```
"""
```

```
Core orchestrator for PlayNAC Kernel
```

```
"""
```

```
from bee.bio_pow import BioPoW
```

```
from earnedpath.simulation_engine import SimulationEngine
```

```
from berc.jas_consensus import JASConsensus
```

```
# ... import other modules
```

```
class PlayNACKernel:
```

```
    def __init__(self, config):
```

```
        self.config = config
```

```
        self.config.load_env()
```

```
        self.config.validate(["WEB3_RPC_URL", "BEE_SECRET_KEY"])
```

```
        self.bio_pow = BioPoW(secret_key=self.config.get("BEE_SECRET_KEY"))
```

```
        self.sim_engine = SimulationEngine()
```

```
self.consensus = JASConsensus()
```

```
# Initialize other modules: GERP, NBERS, GCF, CARE, GEO, SOMT, Media, Nav
```

```
def run(self) -> None:
```

```
    """Main loop: mine blocks and process tasks"""
```

```
    while True:
```

```
        block = self.mine_block()
```

```
        if block:
```

```
            print(f"Mined block: {block.index}")
```

```
def mine_block(self, max_iter: int = 1000):
```

```
    # Stub: implement mining logic
```

```
    pass
```

```
# File: src/earnedpath/ep_node.py
```

```
"""
```

```
EarnedPath node definitions
```

```
"""
```

```
from enum import Enum
```

```
from typing import List
```

```
class EPState(Enum):
```

```
    LOCKED = 0
```

UNLOCKED = 1

COMPLETED = 2

class EPNode:

def __init__(self, node_id: str, deps: List['EPNode'] = None):

self.node_id = node_id

self.dependencies = deps or []

self.state = EPState.LOCKED

self.result = None

def unlock(self) -> None:

if all(d.state == EPState.COMPLETED for d in self.dependencies):

self.state = EPState.UNLOCKED

def complete(self, result: Any) -> None:

self.state = EPState.COMPLETED

self.result = result

File: src/earnedpath/merit_calculator.py

"""

Merit scoring utility

"""

from typing import List

```
class Action:
```

```
    def __init__(self, value: float, weight: float):
```

```
        self.value = value
```

```
        self.weight = weight
```

```
class MeritCalculator:
```

```
    def calculate_merit(self, actions: List[Action]) -> float:
```

```
        return sum(a.value * a.weight for a in actions)
```

```
# File: src/earnedpath/simulation_engine.py
```

```
"""
```

```
EarnedPath simulation engine
```

```
"""
```

```
from typing import Dict, Any
```

```
class SimulationEngine:
```

```
    def __init__(self):
```

```
        # initialize PERT/CPM structures
```

```
        pass
```

```
    def setup_scenario(self, config: Dict[str, Any]) -> None:
```

```
        # Build simulation nodes
```

```
pass
```

```
def step(self) -> Any:
```

```
    # Advance simulation one step
```

```
    pass
```

```
def report(self) -> Any:
```

```
    # Summarize results
```

```
    pass
```

```
# File: src/gianterp/client.py
```

```
"""
```

```
GiantERP API client
```

```
"""
```

```
import requests
```

```
from typing import Dict, Any
```

```
class ResourceGrid:
```

```
    def __init__(self, region_id: str, capacity: float, forecast: Dict[str, float]):
```

```
        self.region_id = region_id
```

```
        self.capacity = capacity
```

```
        self.forecast = forecast
```

```
class GiantERPClient:

    def __init__(self, base_url: str):

        self.base_url = base_url


    def fetch_grid(self, region_id: str) -> ResourceGrid:

        res = requests.get(f"{self.base_url}/grids/{region_id}")

        data = res.json()

        return ResourceGrid(**data)


    def submit_projection(self, proj_data: Dict[str, Any]) -> Any:

        res = requests.post(f"{self.base_url}/projections", json=proj_data)

        return res.json()


# File: src/bee/scanner.py
"""

AuraScanner stub for EEG capture

"""

import numpy as np


class AuraScanner:

    def capture(self) -> np.ndarray:

        return np.random.rand(256)
```

```
def is_device_connected(self) -> bool:
```

```
    return True
```

```
# File: src/bee/bio_pow.py
```

```
"""
```

```
Bio-PoW: Entropic potential generator
```

```
"""
```

```
import numpy as np
```

```
from .scanner import AuraScanner
```

```
class BioPoW:
```

```
    def __init__(self, secret_key: str, gerp_factor: float = 0.618):
```

```
        self.secret_key = secret_key
```

```
        self.scanner = AuraScanner()
```

```
        self.gerp_factor = gerp_factor
```

```
    def generate_ep(self) -> float:
```

```
        raw = self.scanner.capture()
```

```
        entropy = -np.sum(raw * np.log2(raw + 1e-10))
```

```
        return entropy * self.gerp_factor
```

```
    def validate(self, ep_value: float, target: float, tol: float = 0.01) -> bool:
```

```
        return abs(ep_value - target) < tol
```



```
# File: src/berc/models.py
```

```
"""
```

```
MediaTask and JASLink models
```

```
"""
```

```
from dataclasses import dataclass
```

```
@dataclass
```

```
class MediaTask:
```

```
    id: str
```

```
    input_frame: Any
```

```
    task_type: str
```

```
    nonce: int
```

```
    timestamp: float
```

```
    ep_value: float = 0.0
```

```
@dataclass
```

```
class JASLink:
```

```
    source: str
```

```
    target: str
```

```
    weight: float
```

```
    timestamp: float
```

```
# File: src/berc/jas_consensus.py

"""

JAS Graph consensus

"""

import time

import numpy as np

from .models import JASLink, MediaTask

class JASConsensus:

    def __init__(self, threshold: float = 0.6):

        self.threshold = threshold

        self.links: list[JASLink] = []

    def create_link(self, src: MediaTask, tgt: MediaTask, corr: float) -> JASLink:

        link = JASLink(src.id, tgt.id, corr, time.time())

        self.links.append(link)

        return link

    def validate(self, task_id: str) -> bool:

        ws = [l.weight for l in self.links if l.source == task_id or l.target == task_id]

        return (sum(ws)/len(ws)) >= self.threshold if ws else True
```

```
# File: src/nbers/model.py
```

```
"""
```

```
Neural network economic model stub
```

```
"""
```

```
import tensorflow as tf
```

```
class EconomicModel(tf.keras.Model):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.d1 = tf.keras.layers.Dense(64, activation='relu')
```

```
        self.d2 = tf.keras.layers.Dense(32, activation='relu')
```

```
        self.out = tf.keras.layers.Dense(1)
```

```
    def call(self, inputs):
```

```
        x = self.d1(inputs)
```

```
        x = self.d2(x)
```

```
        return self.out(x)
```

```
# File: src/nbers/engine.py
```

```
"""
```

```
Forecast engine for NBERS
```

```
"""
```

```
import numpy as np
```

```
from .model import EconomicModel
```

```
class ForecastEngine:
```

```
    def __init__(self, model: EconomicModel):
```

```
        self.model = model
```

```
    def predict(self, data: np.ndarray) -> np.ndarray:
```

```
        return self.model(data)
```

```
# File: src/gcf/client.py
```

```
"""
```

```
Gracechain client stub
```

```
"""
```

```
from web3 import Web3
```

```
class GracechainClient:
```

```
    def __init__(self, rpc_url: str, abi: Any, addr: str):
```

```
        self.web3 = Web3(Web3.HTTPProvider(rpc_url))
```

```
        self.contract = self.web3.eth.contract(address=addr, abi=abi)
```

```
    def distribute(self, recipient: str, amount: int) -> Any:
```

```
        tx = self.contract.functions.distributeMeritcoin(recipient, amount).buildTransaction()
```

```
        return self.web3.eth.send_transaction(tx)
```

```
# File: src/gcf/token.py

"""

Meritcoin token interface

"""

class Meritcoin:

    def __init__(self, contract):

        self.contract = contract

    def balance_of(self, address: str) -> int:

        return self.contract.functions.balanceOf(address).call()
```

```
# File: src/care/manager.py

"""

CARE property management

"""

from typing import Dict

class CAREManager:

    def compute_pe(self, domain: str, metrics: Dict[str, float]) -> float:

        # protect vs enrich weighting

        return metrics.get('protect', 0) * 0.7 + metrics.get('enrich', 0) * 0.3
```

```
# File: src/geo/router.py
```

```
"""
```

```
Geo routing for Mandala-VERTECA
```

```
"""
```

```
class GODRouter:
```

```
    def route(self, lat: float, lon: float) -> str:
```

```
        # map coordinates to region ID
```

```
        return f"region_{int(lat)}_{int(lon)}"
```

```
# File: src/geo/remediator.py
```

```
"""
```

```
Non-Punitive Remediation
```

```
"""
```

```
class NPRRemediator:
```

```
    def remediate(self, region_id: str) -> None:
```

```
        # apply remediation protocols
```

```
        pass
```

```
# File: src/somt/recorder.py
```

```
"""
```

```
State recorder for sustainability snapshots
```

```
"""
```

```
import hashlib
```

```
import json
```

```
class StateRecorder:
```

```
    def snapshot(self, data: Dict) -> str:
```

```
        digest = hashlib.sha256(json.dumps(data, sort_keys=True).encode()).hexdigest()
```

```
        return digest
```

```
# File: src/somt/gear.py
```

```
"""
```

```
GEAR client stub
```

```
"""
```

```
class GEARClient:
```

```
    def log(self, record: Dict) -> None:
```

```
        # persist environmental data
```

```
        pass
```

```
# File: src/media/media_processor.py
```

```
"""
```

```
Real-time media processing
```

```
"""
```

```
import cv2
```

```
import numpy as np
```

```
from berc.models import MediaTask
```

```
class MediaProcessor:
```

```
    def __init__(self, threshold: float = 0.07):
```

```
        self.threshold = threshold
```

```
    def calculate_md_complexity(self, frame: np.ndarray) -> float:
```

```
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
        hist = cv2.calcHist([gray], [0], None, [256], [0,256]).flatten()
```

```
        norm = hist / hist.sum()
```

```
        entropy = -np.sum(norm * np.log2(norm + 1e-10))
```

```
        return entropy / 8.0
```

```
    def validate_md_complexity(self, frame: np.ndarray) -> bool:
```

```
        return self.calculate_md_complexity(frame) > self.threshold
```

```
    def process_media_task(self, task: MediaTask) -> np.ndarray:
```

```
        if not self.validate_md_complexity(task.input_frame):
```

```
            raise ValueError("MD complexity failed")
```

```
        # stylize frame
```

```
        return cv2.stylization(task.input_frame, sigma_s=60, sigma_r=0.6)
```

```
# File: src/nav/asr.py
```


"""

ASR client abstraction

"""

from abc import ABC, abstractmethod

class ASRClient(ABC):

@abstractmethod

def transcribe(self, audio: Any) -> str:

pass

File: src/nav/intent_parser.py

"""

Intent parser for voice commands

"""

class IntentParser:

def parse(self, text: str) -> Tuple[str, Dict]:

simple rule-based NLU

if 'resource' in text:

return 'allocate_resource', {}

return 'unknown', {}

File: src/nav/dialogue.py

"""

Dialogue manager for HowWay

"""

class DialogueManager:

def handle(self, intent: str, slots: Dict) -> str:

route to modules based on intent

return f"Handled intent {intent} with slots {slots}"

File: src/nav/mandala.py

"""

Mandala-VERTECA translator

"""

from typing import Tuple

class MandalaTranslator:

SYMBOL_MAP = {

'thumb_palm': ('Δ', 'home'),

'index_mudra': ('Δ', 'back'),

'middle_press': ('▽', 'select'),

'ring_swirl': ('▽', 'menu'),

'pinky_wave': ('♀', 'voice'),

}

```
def translate(self, gesture: str) -> Tuple[str, str]:  
    return self.SYMBOL_MAP.get(gesture, ("", ""))
```

```
# File: src/nav/hfvn.py
```

```
"""
```

```
Green Box HFVN environment
```

```
"""
```

```
class GreenBoxEnvironment:
```

```
    def __init__(self, renderer, audio_engine, translator):
```

```
        self.renderer = renderer
```

```
        self.audio = audio_engine
```

```
        self.translator = translator
```

```
        self.active = False
```

```
    def activate(self): pass
```

```
    def deactivate(self): pass
```

```
    def on_gesture(self, gesture): return self.translator.translate(gesture)
```

```
    def on_voice(self, text, kernel): return kernel.nav.dialogue.handle(text, {})
```

```
# File: src/utils/exceptions.py
```

```
"""
```

```
Custom exceptions
```

"""

class KernelError(Exception): pass

File: src/utils/logger.py

"""

Structured JSON logger

"""

import logging

import json

def get_logger(name: str):

logger = logging.getLogger(name)

handler = logging.StreamHandler()

logger.addHandler(handler)

return logger

File: src/utils/helpers.py

"""

Helper decorators and utilities

"""

import time

from functools import wraps

```
def retry(func=None, retries=3, delay=1.0):  
    def decorator(f):  
        @wraps(f)  
        def wrapper(*args, **kwargs):  
            attempts = retries  
            while attempts:  
                try:  
                    return f(*args, **kwargs)  
                except Exception:  
                    attempts -= 1  
                    time.sleep(delay)  
            return None  
        return wrapper  
    return decorator if func is None else decorator(func)
```

```
def timed_cache(ttl=300):  
    # Placeholder for TTL cache decorator  
    def decorator(f): return f  
    return decorator
```