```
#!/usr/bin/env python3
"""Drug discovery simulation orchestrator built on Golden Turing AI architecture.

This module instantiates a quantum-enhanced, multi-agent drug discovery pipeline
modeled on the Golden Turing AI core and the multi-agent roadmap contained in the
repository.  It integrates lambda-scale invariant geometry, curved spacetime
simulators, and a lightweight LLM interface to coordinate the following agents:

- StructuralAnalysisAgent
- LigandDiscoveryAgent (inverse design + scaffold hopping)
- QuantumSimulationAgent
- SynthesisPlannerAgent
- ScreeningAgent
- SafetyAgent
- IPAgent

Each agent posts structured reports to a shared quantum blackboard.  The
simulation optionally grounds itself in public datasets (RCSB PDB, PubChem,
PatentsView, UniProt) when network access is available, falling back to curated
examples if offline.  Quantum utility scores are derived from the
``kg_scale_invariant_metric`` and ``phase4_entanglement`` modules to ensure the
pipeline remains faithful to the repository's λ-scale invariant principles.
"""
from __future__ import annotations

import argparse
import ast
import asyncio
import copy
import csv
import hashlib
import importlib
import io
import itertools
import json
import logging
import math
import os
import random
import shutil
import statistics
import tarfile
import tempfile
import textwrap
```

```
import time
import types
import zipfile
from datetime import datetime
from collections import defaultdict, deque
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any, Callable, Dict, Iterable, List, Optional, Sequence, Set, Tuple
from urllib import error as urlerror, request as urlrequest

import numpy as np

from kg_scale_invariant_metric import (
    FieldParams,
    GeometryParams,
    build_kg_operator,
    compute_modes,
    integrate_profile,
)
from phase4_entanglement import (
    Params as EntanglementParams,
    build_adjacency,
    build_geometry,
    build_hamiltonian,
    single_particle_entropy_for_cut,
)


LAMBDA_DILATION = float(np.sqrt(6.0) / 2.0)
PHI_CONSTANT = float((1.0 + math.sqrt(5.0)) / 2.0)
DUAL_SCALING_ALPHA = float(np.log(LAMBDA_DILATION) / np.log(PHI_CONSTANT))
DEFAULT_RANDOM_SEED = 1337


SKLEARN_AVAILABLE = importlib.util.find_spec("sklearn") is not None
TORCH_AVAILABLE = importlib.util.find_spec("torch") is not None
PLOTLY_AVAILABLE = importlib.util.find_spec("plotly") is not None
MATPLOTLIB_AVAILABLE = importlib.util.find_spec("matplotlib") is not None
SHAP_AVAILABLE = importlib.util.find_spec("shap") is not None

if SKLEARN_AVAILABLE:
    sklearn_metrics_module = importlib.import_module("sklearn.metrics")
    sklearn_model_selection_module = importlib.import_module("sklearn.model_selection")
    sklearn_ensemble_module = importlib.import_module("sklearn.ensemble")
```

```
    sklearn_linear_module = importlib.import_module("sklearn.linear_model")
    sklearn_neural_module = importlib.import_module("sklearn.neural_network")
    RandomForestRegressor = getattr(sklearn_ensemble_module, "RandomForestRegressor")
    RandomForestClassifier = getattr(sklearn_ensemble_module, "RandomForestClassifier")
    LogisticRegression = getattr(sklearn_linear_module, "LogisticRegression")
    MLPRegressor = getattr(sklearn_neural_module, "MLPRegressor")
    ParameterGrid = getattr(sklearn_model_selection_module, "ParameterGrid")
    roc_auc_score = getattr(sklearn_metrics_module, "roc_auc_score")
    precision_recall_fscore_support = getattr(sklearn_metrics_module,
"precision_recall_fscore_support")
else:  # pragma: no cover - optional dependency
    RandomForestRegressor = None
    RandomForestClassifier = None
    LogisticRegression = None
    MLPRegressor = None
    ParameterGrid = None
    roc_auc_score = None
    precision_recall_fscore_support = None


@dataclass
class LambdaShellDescriptor:
    shell_index: int
    lambdaRadius: float
    lambdaCurvature: float
    lambdaEntropy: float
    lambdaEnergyDensity: float
    lambdaBhattacharyya: float
    lambdaOccupancy: float
    lambdaLeakage: float

    def to_dict(self) -> Dict[str, Any]:
        return {
            "shellIndex": self.shell_index,
            "lambdaRadius": self.lambdaRadius,
            "lambdaCurvature": self.lambdaCurvature,
            "lambdaEntropy": self.lambdaEntropy,
            "lambdaEnergyDensity": self.lambdaEnergyDensity,
            "lambdaBhattacharyya": self.lambdaBhattacharyya,
            "lambdaOccupancy": self.lambdaOccupancy,
            "lambdaLeakage": self.lambdaLeakage,
        }
```

```
def set_global_random_seed(seed: int) -> None:
    """Set random seeds across supported libraries for reproducibility."""

    random.seed(seed)
    np.random.seed(seed)
    os.environ.setdefault("PYTHONHASHSEED", str(seed))
    if TORCH_AVAILABLE:
        torch_module = importlib.import_module("torch")
        torch_module.manual_seed(seed)
        if hasattr(torch_module, "cuda") and hasattr(torch_module.cuda, "manual_seed_all"):
            torch_module.cuda.manual_seed_all(seed)
        if hasattr(torch_module, "backends") and hasattr(torch_module.backends, "cudnn"):
            cudnn = torch_module.backends.cudnn
            setattr(cudnn, "deterministic", True)
            setattr(cudnn, "benchmark", False)


# ---------------------------------------------------------------------------
# Lambda training diagnostics hook
# ---------------------------------------------------------------------------


def lambda_shell_training_hook(
    agent_name: str,
    context: "QuantumContext",
    current_state: Dict[str, Any],
) -> Dict[str, Any]:
    """Record shell-aware training diagnostics for downstream learning."""

    descriptors: Iterable[Dict[str, Any]] = ()
    if isinstance(current_state, dict):
        if "descriptors" in current_state:
            descriptors = current_state["descriptors"]
        elif "lambdaShellDiagnostics" in current_state:
            descriptors = current_state["lambdaShellDiagnostics"].get("descriptors", [])
    descriptors = list(descriptors) or context.lambda_shells
    if not descriptors:
        return {
            "agent": agent_name,
            "shellCount": 0,
            "entropyPerShell": [],
            "curvatureGradient": 0.0,
            "lambdaLeakage": 0.0,
            "timestamp": time.time(),
```

```
        }
    entropies = [float(entry.get("lambdaEntropy", 0.0)) for entry in descriptors]
    curvatures = [float(entry.get("lambdaCurvature", 0.0)) for entry in descriptors]
    leakages = [float(entry.get("lambdaLeakage", 0.0)) for entry in descriptors]
    curvature_gradient = float(curvatures[-1] - curvatures[0]) if len(curvatures) > 1 else 0.0
    record = {
        "agent": agent_name,
        "shellCount": len(descriptors),
        "entropyPerShell": entropies,
        "curvatureGradient": curvature_gradient,
        "lambdaLeakage": float(np.mean(leakages)),
        "timestamp": time.time(),
    }
    return record


# ---------------------------------------------------------------------------
# Utility loaders for repository components
# ---------------------------------------------------------------------------

def _dynamic_import(module_name: str, file_name: str):
    """Import repository modules that use non-standard suffixes (e.g. .py.txt)."""
    candidate = Path(file_name)
    if not candidate.exists():
        raise FileNotFoundError(f"Required module '{file_name}' not found")
    text = candidate.read_text(encoding="utf-8", errors="ignore")
    text = text.lstrip("\ufeff")
    text = text.encode("ascii", "ignore").decode("ascii")
    module = types.ModuleType(module_name)
    module.__file__ = str(candidate.resolve())
    exec(compile(text, module.__file__, "exec"), module.__dict__)
    return module


def _extract_default_config(candidate: Path) -> Dict[str, Any]:
    text = candidate.read_text(encoding="utf-8", errors="ignore")
    marker = "DEFAULT_AI_CONFIG"
    idx = text.find(marker)
    if idx == -1:
        return {"ai_state_dim": 32, "max_memory_size": 500}
    brace_start = text.find("{", idx)
    if brace_start == -1:
        return {"ai_state_dim": 32, "max_memory_size": 500}
    depth = 0
```

```
    end_idx = brace_start
    for pos in range(brace_start, len(text)):
        char = text[pos]
        if char == "{":
            depth += 1
        elif char == "}":
            depth -= 1
            if depth == 0:
                end_idx = pos + 1
                break
    snippet = text[brace_start:end_idx]
    try:
        config = ast.literal_eval(snippet)
        if isinstance(config, dict):
            return config
    except Exception:
        pass
    return {"ai_state_dim": 32, "max_memory_size": 500}


def load_golden_turing_ai():
    candidate = Path("golden_turing_module_10.py.txt")
    try:
        module = _dynamic_import("golden_turing_module_10", candidate.name)
        if not hasattr(module, "GoldenTuringAI"):
            raise AttributeError("GoldenTuringAI class not found in the loaded module")
        return module.GoldenTuringAI
    except Exception:
        base_config = _extract_default_config(candidate)

#!/usr/bin/env python3
"""Golden Turing AI Core for Drug Discovery Simulation.

This module implements the full Golden Turing AI with quantum-inspired feedback loops,
multi-agent planning, and integration with the drug discovery pipeline.
"""

import os
import time
import random
import json
import math
import copy
from collections import deque
```

```
from pathlib import Path
from typing import Any, Dict, List, Optional, Tuple
import numpy as np
from decimal import Decimal, getcontext, ROUND_HALF_UP

# Set high precision for awareness calculations
AWARENESS_PREC = 50
getcontext().prec = AWARENESS_PREC
DECIMAL_EPSILON = Decimal('1e-30')


class GoldenTuringAI:
    """Core Golden Turing AI class with quantum-inspired feedback loops for drug discovery."""

    # Action types
    ACTION_ANALYZE_STATE = "ANALYZE_STATE"
    ACTION_ANALYZE_MEMORY = "ANALYZE_MEMORY"
    ACTION_MUTATE_PARAMS = "MUTATE_PARAMS"
    ACTION_BLEND_STATE = "BLEND_STATE"
    ACTION_ADJUST_ANALYSIS_WEIGHTS = "ADJUST_ANALYSIS_WEIGHTS"
    ACTION_ADAPT_RULE_THRESHOLDS = "ADAPT_RULE_THRESHOLDS"
    ACTION_RUN_SIMULATION = "RUN_SIMULATION"
    ACTION_QUANTUM_TUNNEL = "QUANTUM_TUNNEL"
    ACTION_TUNE_ANALYSIS = "TUNE_ANALYSIS"
    ACTION_EXECUTE_PLAN = "EXECUTE_PLAN"

    def __init__(
        self,
        config: Optional[Dict[str, Any]] = None,
        crawler_id: str = "AI",
        initial_state_dim: Optional[int] = None,
    ):
        self.crawler_id = crawler_id

        # Default configuration
        self.config = {
            "ai_state_dim": 64,
            "max_memory_size": 500,
            "learning_rate_base": Decimal("0.05"),
            "learning_rate_decay": Decimal("0.999"),
            "min_learning_rate": Decimal("0.001"),
            "awareness_gain_success": Decimal("0.020"),
            "awareness_gain_analysis": Decimal("0.010"),
            "awareness_gain_simulation_success": Decimal("0.030"),
```

"awareness_gain_tunneling": Decimal("0.15"),
"awareness_gain_meta_analysis": Decimal("0.025"),
"awareness_loss_failure": Decimal("-0.030"),
"awareness_decay": Decimal("0.9997"),
"min_awareness_for_complex_actions": Decimal("0.3"),
"min_awareness_for_simulation": Decimal("0.5"),
"min_awareness_for_mutation": Decimal("0.6"),
"min_awareness_for_tunneling": Decimal("0.7"),
"min_awareness_for_meta_analysis": Decimal("0.65"),
"state_blend_factor": Decimal("0.1"),
"analysis_weight_adapt_rate": Decimal("0.01"),
"rule_threshold_adapt_rate": Decimal("0.005"),
"state_vector_decay": Decimal("0.95"),
"state_vector_noise": Decimal("0.01"),
"simulation_iterations": 50,
"simulation_challenge_levels": 5,
"simulation_multiverse_params": 3,
"simulation_param_variation_scale": Decimal("0.1"),
"quantum_potential_decay": Decimal("0.99"),
"quantum_potential_gain": Decimal("0.01"),
"potential_awareness_boost_factor": Decimal("0.1"),
"entanglement_awareness_threshold": Decimal("0.05"),
"entanglement_param_boost_factor": Decimal("1.5"),
"tunneling_stagnation_threshold": 50,
"tunneling_stagnation_gain_limit": Decimal("0.005"),
"tunneling_resonance_score_threshold": Decimal("0.05"),
"tunneling_potential_threshold": Decimal("0.3"),
"tunneling_state_shift_factor": Decimal("0.3"),
"annealing_awareness_stability_window": 20,
"annealing_stability_threshold": Decimal("0.01"),
"annealing_mutation_scale_factor_stable": Decimal("0.5"),
"annealing_mutation_scale_factor_unstable": Decimal("1.2"),
"annealing_lr_factor_stable": Decimal("0.9"),
"annealing_lr_factor_unstable": Decimal("1.1"),
"interference_blend_factor_scale": Decimal("0.5"),
"meta_analysis_tuning_rate": Decimal("0.005"),
"meta_analysis_weight_noise": Decimal("0.02"),
"meta_analysis_threshold_noise": Decimal("0.01"),
"memory_fidelity_noise_factor": Decimal("0.005"),
"zeno_effect_trigger_count": 5,
"zeno_effect_time_window_sec": 60,
"zeno_effect_awareness_change_threshold": Decimal("0.001"),
"zeno_effect_dampening_factor": Decimal("0.1"),
"save_interval_updates": 10,

```
    "default_analysis_weights": {
        "success_impact": Decimal("1.0"),
        "error_impact": Decimal("-1.5"),
        "prediction_error_impact": Decimal("-0.5"),
        "timing_penalty_factor": Decimal("-0.01"),
        "analysis_bonus": Decimal("0.2"),
        "simulation_analysis_bonus": Decimal("0.3"),
        "tunneling_bonus": Decimal("0.5"),
        "meta_analysis_bonus": Decimal("0.25"),
    },
    "default_rule_thresholds": {
        "action_complexity_awareness": Decimal("0.3"),
        "simulation_trigger_awareness": Decimal("0.5"),
        "mutation_trigger_awareness": Decimal("0.6"),
        "tunneling_trigger_awareness": Decimal("0.7"),
        "meta_analysis_trigger_awareness": Decimal("0.65"),
        "max_recent_observations_for_analysis": 50,
        "resonance_score_threshold_for_tunneling": Decimal("0.05"),
    },
}

# Update with provided config
if config:
    for key, value in config.items():
        if key in self.config and isinstance(self.config[key], dict) and isinstance(value, dict):
            self.config[key].update(value)
        else:
            if isinstance(value, (int, float)):
                self.config[key] = Decimal(str(value))
            else:
                self.config[key] = value

# Set state dimension
if initial_state_dim:
    self.config["ai_state_dim"] = initial_state_dim
self.state_dim = self.config["ai_state_dim"]

# Initialize state
self.state = np.zeros(self.state_dim, dtype=float)

# Memory systems
mem_size = self.config["max_memory_size"]
history_len = max(
    mem_size,
```

```
            self.config["tunneling_stagnation_threshold"],
            self.config["annealing_awareness_stability_window"]
        )

        self.self_memory = deque(maxlen=mem_size)
        self.awareness_history = deque(maxlen=history_len)
        self.action_history = deque(maxlen=self.config["zeno_effect_trigger_count"] * 2)

        # Learning parameters
        self.learning_rate = self.config["learning_rate_base"]
        self.analysis_weights = self.config["default_analysis_weights"].copy()
        self.rule_thresholds = self.config["default_rule_thresholds"].copy()

        # Core metrics
        self.self_awareness = Decimal("0.1")
        self.state_potential = Decimal("0.1")
        self.state_resonance = Decimal("0.0")

        # Multi-agent planning system
        self.planning_stack = deque()
        self.agent_pool = {}
        self.strategy_history = deque(maxlen=50)
        self.blackboard = None

        # Quantum-inspired features
        self.entanglement_records = []
        self.annealing_records = []
        self.tunneling_records = []
        self.blend_records = []
        self.simulation_reflections = []
        self.meta_tuning_records = []
        self.memory_noise_records = []

        # Performance tracking
        self.stagnation_counter = 0
        self.update_counter = 0
        self.last_save_time = time.time()

        # Initialize state with noise
        self._initialize_state()

    def _initialize_state(self):
        """Initialize the AI state vector with random noise."""
        self.state = np.random.normal(0, 0.1, self.state_dim).astype(float)
```

```
def register_agent(self, name: str, agent: Any) -> None:
    """Register a collaborating agent."""
    self.agent_pool[name] = agent

def inject_blackboard_interface(self, blackboard: Any) -> None:
    """Inject blackboard access into the core AI."""
    self.blackboard = blackboard

def prioritize_planning_task(self, task: Dict[str, Any]) -> None:
    """Add a task to the planning stack with high priority."""
    self.planning_stack.appendleft(task)

def enqueue_planning_task(self, task: Dict[str, Any]) -> None:
    """Add a task to the end of the planning stack."""
    self.planning_stack.append(task)

def manage_planning_stack(self) -> Optional[Dict[str, Any]]:
    """Check and execute planned tasks if present."""
    if not self.planning_stack:
        return None
    plan_task = self.planning_stack.popleft()
    return {
        "action_type": self.ACTION_EXECUTE_PLAN,
        "plan_task": plan_task
    }

def delegate_task(self, agent_name: str, task: Dict[str, Any]) -> None:
    """Send a task to a sub-agent."""
    agent = self.agent_pool.get(agent_name)
    if agent and hasattr(agent, "receive_task"):
        agent.receive_task(task)

def update_awareness(self, delta: Decimal) -> Decimal:
    """Update self-awareness with quantum-inspired adjustments."""
    boost = Decimal("1.0")
    if self.state_potential > Decimal("0.5"):
        boost += self.config["potential_awareness_boost_factor"]

    # Check for Zeno effect dampening
    now = time.time()
    recent_actions = [
        (t, a) for t, a in self.action_history
        if now - t < self.config["zeno_effect_time_window_sec"]
```

```
        ]
        if len(recent_actions) >= self.config["zeno_effect_trigger_count"]:
            boost *= self.config["zeno_effect_dampening_factor"]

        adjusted = delta * boost
        self.self_awareness += adjusted
        self.self_awareness = max(Decimal("0.0"), min(Decimal("1.0"), self.self_awareness))
        self.awareness_history.append(float(self.self_awareness))

        return adjusted

    def apply_adaptive_annealing(self) -> Optional[Dict[str, Any]]:
        """Apply adaptive annealing based on awareness stability."""
        if len(self.awareness_history) < 5:
            return None

        window = list(self.awareness_history)[-20:]
        if len(window) <= 1:
            std = Decimal("0.0")
        else:
            std = Decimal(np.std(window).astype(str))

        mode = "exploit" if std <= self.config["annealing_stability_threshold"] else "explore"

        lr_scale = (
            self.config["annealing_lr_factor_stable"]
            if mode == "exploit"
            else self.config["annealing_lr_factor_unstable"]
        )

        mutation_scale = (
            self.config["annealing_mutation_scale_factor_stable"]
            if mode == "exploit"
            else self.config["annealing_mutation_scale_factor_unstable"]
        )

        record = {
            "mode": mode,
            "awarenessStd": float(std),
            "learningRateScale": float(lr_scale),
            "mutationScale": float(mutation_scale),
        }

        self.annealing_records.append(record)
```

```
    if mode == "explore":
        self.mutate_params("explore")

    return record

def perform_quantum_tunnel(self) -> Dict[str, Any]:
    """Perform quantum tunneling to escape local optima."""
    noise = np.random.normal(
        0,
        float(self.config["tunneling_state_shift_factor"]),
        self.state_dim
    )
    self.state = self.state.astype(float) + noise

    payload = {
        "reason": "Awareness stagnation detected; low resonance score in state space.",
        "shift_magnitude": float(self.config["tunneling_state_shift_factor"]),
        "new_hypothesis_directive": "Prioritize scaffold hopping into lambda-toroidal sites.",
    }

    self.tunneling_records.append({
        "newFocus": "lambda-shift",
        "rewardBonus": 0.5,
        "payload": payload
    })

    return payload

def perform_state_blend(self) -> Dict[str, Any]:
    """Blend current state with random interference."""
    random_state = np.random.normal(0, 1.0, self.state_dim)
    current = self.state.astype(float)

    diff_norm = float(np.linalg.norm(current - random_state))
    scale = min(1.0, diff_norm / max(1.0, np.sqrt(random_state.size)))
    blend_factor = float(self.config["state_blend_factor"] * Decimal(str(1.0 + scale)))

    blended = (1 - blend_factor) * current + blend_factor * random_state
    self.state = blended

    payload = {
        "differenceNorm": diff_norm,
        "blendFactor": blend_factor,
```

```
    }

    self.blend_records.append(payload)
    return payload

def mutate_params(self, mode: str) -> None:
    """Mutate analysis parameters based on mode."""
    scale = (
        float(self.config["annealing_mutation_scale_factor_unstable"])
        if mode == "explore"
        else float(self.config["annealing_mutation_scale_factor_stable"])
    )

    # Mutate analysis weights
    for key in self.analysis_weights:
        if isinstance(self.analysis_weights[key], Decimal):
            noise = np.random.normal(0, scale * 0.1)
            new_value = self.analysis_weights[key] + Decimal(str(noise))
            self.analysis_weights[key] = max(Decimal("-1.0"), min(Decimal("2.0"), new_value))

    # Mutate rule thresholds
    for key in self.rule_thresholds:
        if isinstance(self.rule_thresholds[key], Decimal):
            noise = np.random.normal(0, scale * 0.05)
            new_value = self.rule_thresholds[key] + Decimal(str(noise))
            self.rule_thresholds[key] = max(Decimal("0.0"), min(Decimal("1.0"), new_value))

def run_recursive_simulation(self) -> Dict[str, Any]:
    """Run recursive simulation for strategy optimization."""
    variations = []
    for idx in range(self.config["simulation_multiverse_params"]):
        perturb = (idx - 1) * float(self.config["simulation_param_variation_scale"])
        success_rate = float(np.clip(
            0.55 + 0.1 * random.random() + 0.05 * perturb,
            0.0, 1.0
        ))

        variations.append({
            "paramSetId": f"mv-{idx}",
            "ligandWeight": float(np.clip(0.6 + perturb, 0.1, 0.95)),
            "toxicityWeight": float(np.clip(0.3 - perturb, 0.05, 0.9)),
            "simulatedSuccessRate": success_rate,
        })
```

```
    best = max(variations, key=lambda x: x["simulatedSuccessRate"])

    reflection = {
        "variations": variations,
        "bestParamSetId": best["paramSetId"],
        "averageSuccessRate": float(np.mean([v["simulatedSuccessRate"] for v in variations])),
        "awarenessBonus": 0.3,
    }

    self.simulation_reflections.append(reflection)
    self.update_awareness(Decimal("0.03"))

    return reflection

def tune_analysis_parameters(self, avg_reward: float) -> Dict[str, Any]:
    """Tune analysis parameters based on recent performance."""
    avg_reward_dec = Decimal(str(avg_reward))

    meta = {
        "analysisWeights": {
            "prediction_error_impact": float(np.clip(
                1.0 + float(avg_reward_dec), 0.5, 1.5
            )),
            "timing_penalty_factor": float(np.clip(
                0.8 - float(avg_reward_dec) * 0.1, 0.2, 1.2
            )),
        },
        "ruleThresholds": {
            "simulation_trigger_awareness": float(np.clip(
                float(self.self_awareness) - 0.05, 0.1, 0.9
            )),
        },
    }

    self.meta_tuning_records.append(meta)
    return meta

def compile_summary(self) -> Dict[str, Any]:
    """Compile a summary of the AI's current state and history."""
    if len(self.awareness_history) > 1:
        awareness_std = float(np.std(list(self.awareness_history)[-20:]))
    else:
        awareness_std = 0.0
```

```
        return {
            "state": {
                "statePotential": float(self.state_potential),
                "selfAwareness": float(self.self_awareness),
                "stateResonance": float(self.state_resonance),
                "awarenessStd": awareness_std,
                "awarenessHistoryTail": list(self.awareness_history)[-10:],
            },
            "entanglement": self.entanglement_records[-5:] if self.entanglement_records else [],
            "annealing": self.annealing_records[-5:] if self.annealing_records else [],
            "tunneling": self.tunneling_records[-5:] if self.tunneling_records else [],
            "interference": self.blend_records[-5:] if self.blend_records else [],
            "simulations": self.simulation_reflections[-3:] if self.simulation_reflections else [],
            "metaAnalysis": self.meta_tuning_records[-3:] if self.meta_tuning_records else [],
            "memory": self.memory_noise_records[-3:] if self.memory_noise_records else [],
        }


# ---------------------------------------------------------------------------
# Shared data structures
# ---------------------------------------------------------------------------


@dataclass
class QuantumBlackboard:
    """Minimal quantum-inspired blackboard with λ-scale persistence."""

    posts: Dict[str, Dict[str, Any]] = field(default_factory=dict)
    timeline: List[Tuple[str, str]] = field(default_factory=list)
    canonical_basis: str = "phi"

    @staticmethod
    def _lambda_to_phi(value: float) -> float:
        return float(np.sign(value) * np.power(abs(value) + 1e-9, DUAL_SCALING_ALPHA))

    def _canonicalize_scaling(self, payload: Any) -> Any:
        if isinstance(payload, dict):
            normalized: Dict[str, Any] = {}
            basis = payload.get("scalingBasis") or payload.get("basis")
            for key, value in payload.items():
                normalized[key] = self._canonicalize_scaling(value)
            if "descriptors" in payload and payload.get("descriptors") and
isinstance(payload["descriptors"], list):
                descriptors = payload["descriptors"]
```

```
            if descriptors and any("lambda" in k for k in descriptors[0].keys()):
                phi_descriptors = []
                for descriptor in descriptors:
                    phi_descriptors.append(
                        {
                            "phiRadius": self._lambda_to_phi(descriptor.get("lambdaRadius", 0.0)),
                            "phiCurvature": self._lambda_to_phi(descriptor.get("lambdaCurvature",
0.0)),
                            "phiEntropy": self._lambda_to_phi(descriptor.get("lambdaEntropy", 0.0)),
                            "phiEnergyDensity":
self._lambda_to_phi(descriptor.get("lambdaEnergyDensity", 0.0)),
                            "phiBhattacharyya":
self._lambda_to_phi(descriptor.get("lambdaBhattacharyya", 0.0)),
                            "phiOccupancy": self._lambda_to_phi(descriptor.get("lambdaOccupancy",
0.0)),
                            "phiLeakage": self._lambda_to_phi(descriptor.get("lambdaLeakage", 0.0)),
                            "shellIndex": descriptor.get("shellIndex"),
                        }
                    )
                fingerprint = payload.get("summary", {}).get("lambdaShellFingerprint") or
payload.get("fingerprint", [])
                phi_fingerprint = [self._lambda_to_phi(val) for val in fingerprint]
                normalized["canonicalScaling"] = {
                    "basis": self.canonical_basis,
                    "descriptors": phi_descriptors,
                    "fingerprint": phi_fingerprint,
                }
                summary = normalized.setdefault("summary", {})
                summary.setdefault("scalingBasis", basis or "lambda")
                normalized["canonicalScaling"]["summary"] = {"scalingBasis": self.canonical_basis}
            if basis and basis != self.canonical_basis:
                normalized["canonicalBasis"] = self.canonical_basis
            return normalized
        if isinstance(payload, list):
            return [self._canonicalize_scaling(item) for item in payload]
        return payload

    async def post(self, channel: str, report: Dict[str, Any]) -> None:
        canonical_report = self._canonicalize_scaling(copy.deepcopy(report))
        self.posts[channel] = canonical_report
        self.timeline.append((channel, canonical_report.get("reportId") or
canonical_report.get("jobId", "?")))

    async def read(self, channel: str) -> Optional[Dict[str, Any]]:
```

```
        return self.posts.get(channel)

    async def snapshot(self) -> Dict[str, Any]:
        return {"timeline": list(self.timeline), "posts": dict(self.posts)}


@dataclass
class QuantumContext:
    """Holds quantum enhancement metrics shared across agents."""

    curvature_profile: List[float]
    lambda_modes: List[float]
    entanglement_entropy: float
    enhancement_factor: float
    lambda_shells: List[Dict[str, Any]] = field(default_factory=list)
    lambda_basis: Dict[str, Any] = field(default_factory=dict)

    def to_snapshot(self) -> Dict[str, Any]:
        return {
            "curvature_profile": list(self.curvature_profile),
            "lambda_modes": list(self.lambda_modes),
            "entanglement_entropy": float(self.entanglement_entropy),
            "enhancement_factor": float(self.enhancement_factor),
            "lambda_shells": copy.deepcopy(self.lambda_shells),
            "lambda_basis": copy.deepcopy(self.lambda_basis),
        }

    def clone(self) -> "QuantumContext":
        return QuantumContext.from_snapshot(self.to_snapshot())

    @staticmethod
    def from_snapshot(snapshot: Dict[str, Any]) -> "QuantumContext":
        return QuantumContext(
            curvature_profile=list(snapshot.get("curvature_profile", [])),
            lambda_modes=list(snapshot.get("lambda_modes", [])),
            entanglement_entropy=float(snapshot.get("entanglement_entropy", 0.0)),
            enhancement_factor=float(snapshot.get("enhancement_factor", 0.0)),
            lambda_shells=copy.deepcopy(snapshot.get("lambda_shells", [])),
            lambda_basis=copy.deepcopy(snapshot.get("lambda_basis", {})),
        )

    @staticmethod
    def shell_statistics(descriptors: Iterable[Dict[str, Any]]) -> Dict[str, Any]:
        entries = list(descriptors or [])
```

```
        if not entries:
            return {
                "shellCount": 0,
                "entropyMean": 0.0,
                "entropyStd": 0.0,
                "bhattacharyyaMean": 0.0,
                "curvatureMean": 0.0,
                "entropyDistribution": [],
                "occupancyDistribution": [],
            }
        entropies = np.array([float(item.get("lambdaEntropy", 0.0)) for item in entries], dtype=float)
        curvatures = np.array([float(item.get("lambdaCurvature", 0.0)) for item in entries],
dtype=float)
        bhattacharyya = np.array([float(item.get("lambdaBhattacharyya", 0.0)) for item in entries],
dtype=float)
        occupancy = np.array([float(item.get("lambdaOccupancy", 0.0)) for item in entries],
dtype=float)
        entropy_sum = float(np.sum(entropies))
        occ_sum = float(np.sum(occupancy))
        if entropy_sum <= 0.0:
            entropy_dist = np.full(len(entries), 1.0 / float(len(entries)), dtype=float)
        else:
            entropy_dist = entropies / entropy_sum
        if occ_sum <= 0.0:
            occupancy_dist = np.full(len(entries), 1.0 / float(len(entries)), dtype=float)
        else:
            occupancy_dist = occupancy / occ_sum
        return {
            "shellCount": int(len(entries)),
            "entropyMean": float(np.mean(entropies)),
            "entropyStd": float(np.std(entropies)),
            "bhattacharyyaMean": float(np.mean(bhattacharyya)),
            "curvatureMean": float(np.mean(curvatures)),
            "entropyDistribution": entropy_dist.tolist(),
            "occupancyDistribution": occupancy_dist.tolist(),
        }

    @staticmethod
    def diff_snapshots(before: Dict[str, Any], after: Dict[str, Any]) -> Dict[str, Any]:
        before_stats = QuantumContext.shell_statistics(before.get("lambda_shells", []))
        after_stats = QuantumContext.shell_statistics(after.get("lambda_shells", []))
        max_len = max(len(before_stats["occupancyDistribution"]),
len(after_stats["occupancyDistribution"]))
```

```
        def _pad(values: List[float]) -> np.ndarray:
            arr = np.asarray(values, dtype=float)
            if arr.size < max_len:
                arr = np.pad(arr, (0, max_len - arr.size), constant_values=0.0)
            return arr

        before_occ = _pad(before_stats["occupancyDistribution"])
        after_occ = _pad(after_stats["occupancyDistribution"])
        occupancy_l1 = float(np.sum(np.abs(after_occ - before_occ)))
        basis_before = before.get("lambda_basis", {})
        basis_after = after.get("lambda_basis", {})
        basis_keys = set(basis_before.keys()) | set(basis_after.keys())
        basis_changes: Dict[str, Dict[str, Any]] = {}
        for key in basis_keys:
            if basis_before.get(key) != basis_after.get(key):
                basis_changes[key] = {"before": basis_before.get(key), "after": basis_after.get(key)}
        return {
            "shellCountDelta": after_stats["shellCount"] - before_stats["shellCount"],
            "entropyMeanDelta": after_stats["entropyMean"] - before_stats["entropyMean"],
            "entropyStdDelta": after_stats["entropyStd"] - before_stats["entropyStd"],
            "bhattacharyyaDelta": after_stats["bhattacharyyaMean"] -
before_stats["bhattacharyyaMean"],
            "curvatureMeanDelta": after_stats["curvatureMean"] - before_stats["curvatureMean"],
            "occupancyL1Delta": occupancy_l1,
            "entanglementEntropyDelta": float(after.get("entanglement_entropy", 0.0) -
before.get("entanglement_entropy", 0.0)),
            "enhancementFactorDelta": float(after.get("enhancement_factor", 0.0) -
before.get("enhancement_factor", 0.0)),
            "lambdaBasisChanges": basis_changes,
        }


# ---------------------------------------------------------------------
# Lightweight LLM interface
# ---------------------------------------------------------------------


class LightweightLLM:
    """Adapter for the TinyLlama GGUF model with graceful degradation."""

    def __init__(self, model_path: Path, max_tokens: int = 256):
        self.model_path = model_path
        self.max_tokens = max_tokens
        self._llama = None
```

```
        try:
            from llama_cpp import Llama  # type: ignore

            if model_path.exists():
                self._llama = Llama(model_path=str(model_path), n_ctx=2048,
n_threads=os.cpu_count() or 4)
        except Exception:
            # llama-cpp-python not available; the interface will synthesize responses heuristically
            self._llama = None

    def complete(self, prompt: str, temperature: float = 0.2) -> str:
        if self._llama is not None:
            output = self._llama(prompt=prompt, max_tokens=self.max_tokens,
temperature=temperature, stop=["###"])
            if isinstance(output, dict):
                choices = output.get("choices", [])
                if choices:
                    return choices[0].get("text", "").strip()
        # Fallback heuristic completion
        summary = textwrap.shorten(prompt.split("\n")[-1], width=200)
        return f"Heuristic plan based on context: {summary}"


# ---------------------------------------------------------------------------
# Machine learning integration utilities
# ---------------------------------------------------------------------------


class FeatureExtractor:
    """Feature extraction for diverse chemical, structural, and quantum data."""

    def __init__(self, fingerprint_size: int = 256) -> None:
        self.fingerprint_size = fingerprint_size

    def _hash_sequence(self, sequence: str, size: Optional[int] = None) -> np.ndarray:
        length = size or self.fingerprint_size
        if length <= 0:
            length = 64
        vector = np.zeros(length, dtype=float)
        if not sequence:
            return vector
        for idx, char in enumerate(sequence):
            bucket = (hash((char, idx)) % length + length) % length
            vector[bucket] += 1.0
```

```
    norm = np.linalg.norm(vector)
    if norm > 0:
        vector /= norm
    return vector

def featurize_smiles(self, smiles: str) -> np.ndarray:
    return self._hash_sequence(smiles, self.fingerprint_size)

def featurize_sequence(self, sequence: str) -> np.ndarray:
    return self._hash_sequence(sequence, self.fingerprint_size // 2)

def featurize_quantum_sample(self, sample: Dict[str, Any]) -> np.ndarray:
    orbital = np.array(sample.get("orbitalOccupations", [0.25, 0.25, 0.25, 0.25]), dtype=float)
    entropy = float(sample.get("entanglementEntropy", 0.1))
    binding = float(sample.get("bindingEnergy", -5.0))
    fidelity = float(sample.get("fidelity", 0.95))
    lambda_fp = np.asarray(sample.get("lambdaShellFingerprint", [0.0] * 7), dtype=float)
    descriptor = np.concatenate([
        np.array([binding, entropy, fidelity], dtype=float),
        orbital,
        lambda_fp[:7],
    ])
    return descriptor

def featurize_pocket(self, pocket: Dict[str, Any]) -> np.ndarray:
    properties = pocket.get("properties", {})
    vector = np.array(
        [
            float(pocket.get("druggabilityScore", 0.0)),
            float(properties.get("volume", 1.0)),
            float(properties.get("hydrophobicity", 0.5)),
            float(properties.get("electrostaticPotential", -1.0)),
        ],
        dtype=float,
    )
    return vector

def featurize_lambda_shells(self, descriptors: Iterable[Dict[str, Any]]) -> np.ndarray:
    vector: List[float] = []
    for entry in descriptors:
        vector.extend(
            [
                float(entry.get("lambdaRadius", 0.0)),
                float(entry.get("lambdaCurvature", 0.0)),
```

```
                float(entry.get("lambdaEntropy", 0.0)),
                float(entry.get("lambdaEnergyDensity", 0.0)),
                float(entry.get("lambdaBhattacharyya", 0.0)),
                float(entry.get("lambdaOccupancy", 0.0)),
                float(entry.get("lambdaLeakage", 0.0)),
            ]
        )
    if not vector:
        vector = [0.0] * 7
    return np.array(vector, dtype=float)

def combine_features(self, *features: np.ndarray) -> np.ndarray:
    if not features:
        return np.zeros(1, dtype=float)
    flattened = [np.atleast_1d(feature).astype(float) for feature in features]
    return np.concatenate(flattened)


@dataclass
class BenchmarkDatasetRecord:
    dataset: str
    task: str
    smiles: str
    label: float
    metadata: Dict[str, Any]


class BenchmarkDatasetUtility:
    """Downloads and preprocesses benchmark datasets into unified records."""

    SOURCES: Dict[str, Dict[str, Any]] = {
        "DUD-E": {
            "url":
"https://raw.githubusercontent.com/deepchem/deepchem/master/examples/assets/dude.csv",
            "format": "csv",
            "columns": {"smiles": "smiles", "label": "label"},
        },
        "ChEMBL": {
            "url":
"https://raw.githubusercontent.com/chembl/chembl_webresource_client/master/chembl_webres
ource_client/tests/resources/activity_sample.csv",
            "format": "csv",
            "columns": {"smiles": "canonical_smiles", "label": "standard_value"},
        },
```

```
        "ZINC15": {
            "url":
"https://raw.githubusercontent.com/deepchem/deepchem/master/examples/assets/zinc15_samp
le.csv",
            "format": "csv",
            "columns": {"smiles": "smiles", "label": None},
        },
    }

    def __init__(self, cache_dir: Path | None = None) -> None:
        self.cache_dir = cache_dir or Path("datasets")
        self.cache_dir.mkdir(parents=True, exist_ok=True)

    def _download(self, dataset: str) -> Optional[Path]:
        spec = self.SOURCES.get(dataset)
        if spec is None:
            return None
        target_path = self.cache_dir / f"{dataset.lower()}_raw.{spec['format']}"
        if target_path.exists():
            return target_path
        url = spec.get("url")
        if not url:
            return None
        try:
            with urlrequest.urlopen(url, timeout=30) as response:
                data = response.read()
            target_path.write_bytes(data)
            return target_path
        except Exception as exc:  # pragma: no cover - network best effort
            logging.warning("Failed to download %s dataset: %s", dataset, exc)
            return None

    def _load_csv(
        self,
        dataset: str,
        path: Path,
        columns: Dict[str, Optional[str]],
        limit: int,
    ) -> List[BenchmarkDatasetRecord]:
        records: List[BenchmarkDatasetRecord] = []
        label_column = columns.get("label")
        with path.open("r", encoding="utf-8", errors="ignore") as handle:
            reader = csv.DictReader(handle)
            for idx, row in enumerate(reader):
```

```
            if limit and idx >= limit:
                break
            smiles = row.get(columns.get("smiles", "smiles"), "")
            if not smiles:
                continue
            raw_label = float(row.get(label_column, 0.0)) if label_column else 0.0
            label = raw_label
            if dataset == "ChEMBL":
                label = 1.0 if raw_label and raw_label < 1000 else 0.0
            elif dataset == "ZINC15":
                label = 0.5  # availability only
            metadata = {"sourceRow": idx, "raw": row}
            records.append(
                BenchmarkDatasetRecord(
                    dataset=dataset,
                    task=f"benchmark.{dataset.lower()}",
                    smiles=smiles,
                    label=float(label),
                    metadata=metadata,
                )
            )
    return records

def load(self, dataset: str, limit: int = 500) -> List[BenchmarkDatasetRecord]:
    spec = self.SOURCES.get(dataset)
    if spec is None:
        return []
    path = self._download(dataset)
    if path is not None and spec.get("format") == "csv":
        try:
            return self._load_csv(dataset, path, spec["columns"], limit)
        except Exception as exc:  # pragma: no cover - defensive
            logging.warning("Failed to parse %s dataset: %s", dataset, exc)
    logging.info("Falling back to synthetic %s benchmark records", dataset)
    rng = np.random.default_rng(DEFAULT_RANDOM_SEED)
    synthetic: List[BenchmarkDatasetRecord] = []
    for idx in range(limit):
        smiles = f"C{idx}H{idx}O{idx%3}"
        label = float(rng.random())
        synthetic.append(
            BenchmarkDatasetRecord(
                dataset=dataset,
                task=f"benchmark.{dataset.lower()}",
                smiles=smiles,
```

```
                label=label,
                metadata={"synthetic": True, "index": idx},
            )
        )
    return synthetic

@dataclass
class DatasetSplit:
    train_X: np.ndarray
    train_y: np.ndarray
    val_X: np.ndarray
    val_y: np.ndarray
    test_X: np.ndarray
    test_y: np.ndarray
    normalization: Dict[str, np.ndarray]
    metadata: Dict[str, Any]


class DatasetManager:
    """Curates datasets with normalization, augmentation, and splits."""

    def __init__(
        self,
        feature_extractor: FeatureExtractor,
        benchmark_utility: Optional[BenchmarkDatasetUtility] = None,
        random_seed: int = DEFAULT_RANDOM_SEED,
    ) -> None:
        self.feature_extractor = feature_extractor
        self.records: Dict[str, List[Dict[str, Any]]] = defaultdict(list)
        self.splits: Dict[str, DatasetSplit] = {}
        self.task_types: Dict[str, str] = {}
        self.benchmark_metadata: Dict[str, Any] = {}
        self.random_seed = random_seed
        self.benchmark_utility = benchmark_utility or BenchmarkDatasetUtility()

    def register_record(self, task: str, features: np.ndarray, label: float, metadata: Dict[str, Any]) ->
None:
        self.records[task].append({
            "features": np.asarray(features, dtype=float),
            "label": float(label),
            "metadata": metadata,
        })
        task_type = self.task_types.get(task)
        inferred = "classification" if float(label).is_integer() and label in (0.0, 1.0) else "regression"
```

```
        if task_type is None:
            self.task_types[task] = inferred
        elif task_type != inferred:
            self.task_types[task] = "mixed"

    def _normalize(self, X: np.ndarray) -> Tuple[np.ndarray, Dict[str, np.ndarray]]:
        if X.size == 0:
            return X, {"mean": np.zeros(1), "std": np.ones(1)}
        mean = X.mean(axis=0)
        std = X.std(axis=0)
        std[std == 0] = 1.0
        normalized = (X - mean) / std
        return normalized, {"mean": mean, "std": std}

    def build_split(self, task: str, test_ratio: float = 0.2, val_ratio: float = 0.2) -> DatasetSplit:
        records = self.records.get(task, [])
        if not records:
            empty = np.zeros((0, 4))
            return DatasetSplit(empty, empty, empty, empty, empty, empty, {"mean": np.zeros(1),
"std": np.ones(1)}, {"records": 0})
        features = np.stack([entry["features"] for entry in records])
        labels = np.array([entry["label"] for entry in records], dtype=float)
        idx = np.arange(len(records))
        rng = np.random.default_rng(self.random_seed)
        rng.shuffle(idx)
        features = features[idx]
        labels = labels[idx]
        n_test = max(1, int(len(records) * test_ratio)) if len(records) > 3 else max(0, len(records) -
2)
        n_val = max(1, int(len(records) * val_ratio)) if len(records) > 3 else 1
        n_train = max(1, len(records) - n_test - n_val)
        test_X, test_y = features[:n_test], labels[:n_test]
        val_X, val_y = features[n_test:n_test + n_val], labels[n_test:n_test + n_val]
        train_X, train_y = features[n_test + n_val:], labels[n_test + n_val:]
        norm_train_X, normalization = self._normalize(train_X)
        norm_val_X = (val_X - normalization["mean"]) / normalization["std"] if val_X.size else val_X
        norm_test_X = (test_X - normalization["mean"]) / normalization["std"] if test_X.size else
test_X
        split = DatasetSplit(
            norm_train_X,
            train_y,
            norm_val_X,
            val_y,
            norm_test_X,
```

```
            test_y,
            normalization,
            {"records": len(records)},
        )
        self.splits[task] = split
        return split

    def get_split(self, task: str) -> Optional[DatasetSplit]:
        return self.splits.get(task)

    def get_task_type(self, task: str) -> str:
        return self.task_types.get(task, "regression")

    def augment_with_quantum_samples(self, task: str, samples: List[Dict[str, Any]], target_key:
str) -> None:
        for sample in samples:
            features = self.feature_extractor.featurize_quantum_sample(sample)
            label = float(sample.get(target_key, 0.0))
            self.register_record(task, features, label, {"source": "quantum_reference", "ligandId":
sample.get("ligandId")})

    def integrate_benchmark_dataset(
        self,
        dataset: str,
        context: QuantumContext,
        limit: int = 500,
    ) -> Dict[str, Any]:
        records = self.benchmark_utility.load(dataset, limit)
        lambda_fp = np.asarray(LambdaScalingToolkit.fingerprint(context.lambda_shells),
dtype=float)
        lambda_shell_features =
self.feature_extractor.featurize_lambda_shells(context.lambda_shells)
        ingested = 0
        last_task = f"benchmark.{dataset.lower()}"
        for record in records:
            smiles_feat = self.feature_extractor.featurize_smiles(record.smiles)
            feature_vec = self.feature_extractor.combine_features(smiles_feat,
lambda_shell_features)
            feature_vec = self.feature_extractor.combine_features(feature_vec, lambda_fp)
            metadata = {"dataset": dataset, **record.metadata}
            self.register_record(record.task, feature_vec, record.label, metadata)
            ingested += 1
            last_task = record.task
        if ingested:
```

```
            split = self.build_split(last_task)
            normalization_summary: Dict[str, Any] = {}
            if split and isinstance(split.normalization, dict):
                normalization_summary = {
                    key: np.asarray(value).tolist()
                    for key, value in split.normalization.items()
                }
            self.benchmark_metadata[dataset] = {
                "records": ingested,
                "task": last_task,
                "taskType": self.get_task_type(last_task),
                "normalization": normalization_summary,
            }
        else:
            self.benchmark_metadata[dataset] = {"records": 0, "task":
f"benchmark.{dataset.lower()}"}
        return self.benchmark_metadata[dataset]

    def prepare_benchmarks(
        self,
        datasets: Sequence[str],
        context: QuantumContext,
        limit: int = 500,
    ) -> Dict[str, Any]:
        summary: Dict[str, Any] = {}
        for name in datasets:
            summary[name] = self.integrate_benchmark_dataset(name, context, limit=limit)
        return summary


class StatisticalValidationEngine:
    """Computes statistical metrics and manages validation history."""

    def __init__(self, random_seed: int = DEFAULT_RANDOM_SEED) -> None:
        self.random_seed = random_seed
        self.history: List[Dict[str, Any]] = []

    @staticmethod
    def _bhattacharyya_divergence(p: np.ndarray, q: np.ndarray) -> float:
        p_sum = float(np.sum(p))
        q_sum = float(np.sum(q))
        if p_sum == 0 or q_sum == 0:
            return float("inf")
        p_norm = p / p_sum
```

```
        q_norm = q / q_sum
        coefficient = float(np.sum(np.sqrt(p_norm * q_norm)))
        coefficient = float(np.clip(coefficient, 1e-9, 1.0))
        return float(-math.log(coefficient))

    @staticmethod
    def _simple_auc(y_true: np.ndarray, scores: np.ndarray) -> float:
        order = np.argsort(scores)
        y_sorted = y_true[order]
        scores_sorted = scores[order]
        cum_pos = np.cumsum(y_sorted[::-1])[::-1]
        cum_neg = np.cumsum(1 - y_sorted[::-1])[::-1]
        total_pos = cum_pos[0] if cum_pos.size else 0.0
        total_neg = cum_neg[0] if cum_neg.size else 0.0
        if total_pos == 0 or total_neg == 0:
            return 0.5
        tpr = cum_pos / total_pos
        fpr = cum_neg / total_neg
        return float(np.trapz(tpr, fpr))

    def regression_metrics(self, y_true: np.ndarray, y_pred: np.ndarray) -> Dict[str, float]:
        if y_true.size == 0 or y_pred.size == 0:
            return {"mae": 0.0, "rmse": 0.0, "r2": 0.0, "bhattacharyya": 0.0}
        residuals = y_true - y_pred
        mae = float(np.mean(np.abs(residuals))) if residuals.size else float("nan")
        rmse = float(np.sqrt(np.mean(residuals ** 2))) if residuals.size else float("nan")
        if residuals.size:
            numerator = float(np.sum((y_true - y_pred) ** 2))
            denominator = float(np.sum((y_true - np.mean(y_true)) ** 2) + 1e-9)
            r2 = 1.0 - numerator / denominator
        else:
            r2 = float("nan")
        low = float(np.min(y_true)) if y_true.size else 0.0
        high = float(np.max(y_true)) if y_true.size else 1.0
        if math.isclose(low, high):
            high = low + 1.0
        distribution_true = np.histogram(y_true, bins=20, range=(low, high))[0]
        distribution_pred = np.histogram(y_pred, bins=20, range=(low, high))[0]
        bhatt = self._bhattacharyya_divergence(distribution_true.astype(float),
distribution_pred.astype(float))
        metrics = {"mae": mae, "rmse": rmse, "r2": r2, "bhattacharyya": bhatt}
        return metrics

    def classification_metrics(
```

```
        self,
        y_true: np.ndarray,
        y_scores: np.ndarray,
        threshold: float = 0.5,
    ) -> Dict[str, float]:
        if y_true.size == 0:
            return {"accuracy": 0.0, "precision": 0.0, "recall": 0.0, "f1": 0.0, "auc": 0.5,
"bhattacharyya": 0.0}
        probs = np.clip(y_scores, 0.0, 1.0)
        preds = (probs >= threshold).astype(int)
        accuracy = float(np.mean(preds == y_true)) if y_true.size else 0.0
        tp = float(np.sum((preds == 1) & (y_true == 1)))
        fp = float(np.sum((preds == 1) & (y_true == 0)))
        fn = float(np.sum((preds == 0) & (y_true == 1)))
        precision = tp / (tp + fp + 1e-9)
        recall = tp / (tp + fn + 1e-9)
        f1 = 2 * precision * recall / (precision + recall + 1e-9)
        if roc_auc_score is not None:
            auc = float(roc_auc_score(y_true, probs))
        else:
            auc = self._simple_auc(y_true.astype(float), probs.astype(float))
        distribution_true = np.array([np.mean(y_true == 0), np.mean(y_true == 1)], dtype=float)
        distribution_pred = np.array([np.mean(preds == 0), np.mean(preds == 1)], dtype=float)
        if distribution_true.sum() == 0:
            distribution_true = np.array([0.5, 0.5], dtype=float)
        if distribution_pred.sum() == 0:
            distribution_pred = np.array([0.5, 0.5], dtype=float)
        bhatt = self._bhattacharyya_divergence(distribution_true, distribution_pred)
        metrics = {
            "accuracy": accuracy,
            "precision": float(precision),
            "recall": float(recall),
            "f1": float(f1),
            "auc": float(auc),
            "bhattacharyya": float(bhatt),
        }
        if precision_recall_fscore_support is not None:
            precision_arr, recall_arr, f1_arr, _ = precision_recall_fscore_support(
                y_true,
                preds,
                average=None,
                zero_division=0,
            )
            metrics["precisionPerClass"] = precision_arr.tolist()
```

```
        metrics["recallPerClass"] = recall_arr.tolist()
        metrics["f1PerClass"] = f1_arr.tolist()
    return metrics

def k_fold_cross_validation(
    self,
    model_factory: Callable[[], Any],
    X: np.ndarray,
    y: np.ndarray,
    folds: int = 5,
    task_type: str = "regression",
) -> Dict[str, Any]:
    if X.size == 0:
        return {"folds": [], "aggregate": {}}
    rng = np.random.default_rng(self.random_seed)
    indices = np.arange(len(y))
    rng.shuffle(indices)
    fold_indices = np.array_split(indices, max(1, folds))
    fold_results: List[Dict[str, float]] = []
    for fold_id, test_idx in enumerate(fold_indices):
        train_idx = np.setdiff1d(indices, test_idx)
        model = model_factory()
        X_train = X[train_idx]
        y_train = y[train_idx]
        X_test = X[test_idx]
        y_test = y[test_idx]
        if hasattr(model, "fit"):
            model.fit(X_train, y_train)
        elif isinstance(model, MLModelBase):  # pragma: no cover - unlikely branch
            mean = X_train.mean(axis=0) if X_train.size else np.zeros(X.shape[1])
            std = X_train.std(axis=0)
            std[std == 0] = 1.0
            norm_train = (X_train - mean) / std if X_train.size else X_train
            norm_test = (X_test - mean) / std if X_test.size else X_test
            split = DatasetSplit(
                norm_train,
                y_train,
                norm_test,
                y_test,
                norm_test,
                y_test,
                {"mean": mean, "std": std},
                {"records": int(len(y_train))},
            )
```

```
            model.train(split)
        else:
            continue
        if task_type == "classification":
            if hasattr(model, "predict_proba"):
                scores = model.predict_proba(X_test)[:, -1]
            else:
                scores = model.predict(X_test)
            metrics = self.classification_metrics(y_test, scores)
        else:
            preds = model.predict(X_test)
            metrics = self.regression_metrics(y_test, preds)
        metrics["fold"] = fold_id
        fold_results.append(metrics)
    aggregate: Dict[str, float] = {}
    if fold_results:
        keys = fold_results[0].keys()
        for key in keys:
            if key == "fold":
                continue
            values = [entry[key] for entry in fold_results if isinstance(entry.get(key), (int, float))]
            if values:
                aggregate[key] = float(np.mean(values))
    result = {"folds": fold_results, "aggregate": aggregate, "taskType": task_type}
    self.history.append({"type": "kfold", "result": result})
    return result

def compare_model_variants(self, evaluations: Dict[str, Dict[str, Any]]) -> Dict[str, Any]:
    summary: Dict[str, Any] = {"best": {}, "metrics": {}}
    metric_scores: Dict[str, List[Tuple[str, float]]] = defaultdict(list)
    for model_name, metrics in evaluations.items():
        for metric, value in metrics.items():
            if isinstance(value, (int, float)) and not math.isnan(value):
                metric_scores[metric].append((model_name, float(value)))
    for metric, pairs in metric_scores.items():
        if not pairs:
            continue
        best_entry = max(pairs, key=lambda item: item[1])
        summary["best"][metric] = {"model": best_entry[0], "score": best_entry[1]}
        summary["metrics"][metric] = {name: score for name, score in pairs}
    self.history.append({"type": "comparison", "summary": summary})
    return summary

def log_evaluation(self, tag: str, metrics: Dict[str, Any]) -> None:
```

```
        entry = {"tag": tag, "metrics": metrics, "timestamp": time.time()}
        self.history.append(entry)


class HyperparameterOptimizer:
    """Performs grid search and Bayesian-inspired random search for tuning."""

    def __init__(self, validation_engine: StatisticalValidationEngine, random_seed: int =
DEFAULT_RANDOM_SEED) -> None:
        self.validation_engine = validation_engine
        self.random_seed = random_seed
        self.records: List[Dict[str, Any]] = []

    def grid_search(
        self,
        model_factory: Callable[[Dict[str, Any]], Any],
        param_grid: Dict[str, Sequence[Any]],
        split: DatasetSplit,
        task_type: str,
    ) -> Dict[str, Any]:
        best_score = -float("inf")
        best_params: Dict[str, Any] = {}
        best_metrics: Dict[str, Any] = {}
        param_iterator: Iterable[Dict[str, Any]]
        if ParameterGrid is not None:
            param_iterator = ParameterGrid(param_grid)
        else:
            keys = list(param_grid.keys())
            param_iterator = (
                {key: values[idx % len(values)] for idx, key in enumerate(keys)}
                for _ in range(max(1, len(keys)))
            )
        for params in param_iterator:
            model = model_factory(dict(params))
            if hasattr(model, "fit"):
                model.fit(split.train_X, split.train_y)
            elif isinstance(model, MLModelBase):
                model.train(split)
            else:
                continue
            if task_type == "classification":
                if hasattr(model, "predict_proba"):
                    scores = model.predict_proba(split.val_X)[:, -1] if split.val_X.size else
model.predict_proba(split.train_X)[:, -1]
```

```
                else:
                    scores = model.predict(split.val_X if split.val_X.size else split.train_X)
                metrics = self.validation_engine.classification_metrics(split.val_y if split.val_y.size else
split.train_y, scores)
                score = metrics.get("f1", -float("inf"))
            else:
                preds = model.predict(split.val_X) if split.val_X.size else model.predict(split.train_X)
                metrics = self.validation_engine.regression_metrics(split.val_y if split.val_y.size else
split.train_y, preds)
                score = -metrics.get("rmse", float("inf"))
            self.records.append({"method": "grid", "params": dict(params), "metrics": metrics})
            if score > best_score:
                best_score = score
                best_params = dict(params)
                best_metrics = metrics
        return {"bestParams": best_params, "bestMetrics": best_metrics, "method": "grid"}

    def bayesian_search(
        self,
        model_factory: Callable[[Dict[str, Any]], Any],
        param_space: Dict[str, Tuple[float, float]],
        split: DatasetSplit,
        task_type: str,
        iterations: int = 15,
    ) -> Dict[str, Any]:
        rng = np.random.default_rng(self.random_seed)
        best_score = -float("inf")
        best_params: Dict[str, Any] = {}
        best_metrics: Dict[str, Any] = {}
        for _ in range(iterations):
            params = {key: float(rng.uniform(low, high)) for key, (low, high) in param_space.items()}
            model = model_factory(dict(params))
            if hasattr(model, "fit"):
                model.fit(split.train_X, split.train_y)
            elif isinstance(model, MLModelBase):
                model.train(split)
            else:
                continue
            if task_type == "classification":
                if hasattr(model, "predict_proba"):
                    scores = model.predict_proba(split.val_X)[:, -1] if split.val_X.size else
model.predict_proba(split.train_X)[:, -1]
                else:
                    scores = model.predict(split.val_X if split.val_X.size else split.train_X)
```

```
            metrics = self.validation_engine.classification_metrics(split.val_y if split.val_y.size else
split.train_y, scores)
            score = metrics.get("auc", -float("inf"))
        else:
            preds = model.predict(split.val_X) if split.val_X.size else model.predict(split.train_X)
            metrics = self.validation_engine.regression_metrics(split.val_y if split.val_y.size else
split.train_y, preds)
            score = -metrics.get("rmse", float("inf"))
        record = {"method": "bayesian", "params": params, "metrics": metrics}
        self.records.append(record)
        if score > best_score:
            best_score = score
            best_params = dict(params)
            best_metrics = metrics
    return {"bestParams": best_params, "bestMetrics": best_metrics, "method": "bayesian"}


class TrainingVisualizationLogger:
    """Collects metrics and renders charts for quantum training telemetry."""

    def __init__(self, output_dir: Path | None = None) -> None:
        self.output_dir = output_dir or Path("outputs") / "visualizations"
        self.output_dir.mkdir(parents=True, exist_ok=True)
        self.metric_history: Dict[str, List[Dict[str, Any]]] = defaultdict(list)

    def log_metrics(self, series: str, step: int, metrics: Dict[str, float]) -> Dict[str, Any]:
        entry = {"step": step, "metrics": metrics, "timestamp": time.time()}
        self.metric_history[series].append(entry)
        return entry

    def render(self) -> List[str]:
        generated: List[str] = []
        if MATPLOTLIB_AVAILABLE:
            matplotlib_module = importlib.import_module("matplotlib")
            matplotlib_module.use("Agg")
            plt = importlib.import_module("matplotlib.pyplot")
            for series, records in self.metric_history.items():
                if not records:
                    continue
                steps = [entry["step"] for entry in records]
                metrics = records[0]["metrics"].keys()
                fig, ax = plt.subplots(figsize=(8, 4))
                for metric in metrics:
                    values = [entry["metrics"].get(metric, float("nan")) for entry in records]
```

```
                ax.plot(steps, values, label=metric)
            ax.set_title(f"{series} metrics")
            ax.set_xlabel("Step")
            ax.set_ylabel("Value")
            ax.legend()
            fig_path = self.output_dir / f"{series.replace(' ', '_')}.png"
            fig.tight_layout()
            fig.savefig(fig_path)
            plt.close(fig)
            generated.append(str(fig_path))
        elif PLOTLY_AVAILABLE:  # pragma: no cover - optional
            plotly_module = importlib.import_module("plotly.graph_objects")
            for series, records in self.metric_history.items():
                if not records:
                    continue
                steps = [entry["step"] for entry in records]
                fig = plotly_module.Figure()
                for metric in records[0]["metrics"].keys():
                    values = [entry["metrics"].get(metric, float("nan")) for entry in records]
                    fig.add_trace(plotly_module.Scatter(x=steps, y=values, mode="lines",
name=metric))
                fig.update_layout(title=f"{series} metrics", xaxis_title="Step", yaxis_title="Value")
                fig_path = self.output_dir / f"{series.replace(' ', '_')}.html"
                fig.write_html(fig_path)
                generated.append(str(fig_path))
        return generated


class ExplainabilityEngine:
    """Generates feature attributions for lambda-aware models."""

    def __init__(self, random_seed: int = DEFAULT_RANDOM_SEED) -> None:
        self.random_seed = random_seed

    def explain(
        self,
        model: Any,
        features: np.ndarray,
        task_type: str,
        shell_size: int = 7,
    ) -> Dict[str, Any]:
        feature_array = np.atleast_2d(np.asarray(features, dtype=float))
        if feature_array.size == 0:
            return {"importance": [], "shellAttention": [], "method": "empty"}
```

```
importance: np.ndarray
method = "permutation"
if SHAP_AVAILABLE:
    shap_module = importlib.import_module("shap")
    try:
        explainer = shap_module.Explainer(lambda x: model.predict(x))
        shap_values = explainer(feature_array)
        importance = np.mean(np.abs(shap_values.values), axis=0)
        method = "shap"
    except Exception:
        importance = np.zeros(feature_array.shape[1])
elif hasattr(model, "feature_importances_"):
    importance = np.asarray(model.feature_importances_, dtype=float)
    method = "feature_importances"
elif hasattr(model, "coef_"):
    importance = np.abs(np.asarray(model.coef_).reshape(-1))
    method = "coefficients"
else:
    rng = np.random.default_rng(self.random_seed)
    baseline_pred = model.predict(feature_array)
    importance = np.zeros(feature_array.shape[1])
    for idx in range(feature_array.shape[1]):
        perturbed = feature_array.copy()
        rng.shuffle(perturbed[:, idx])
        perturbed_pred = model.predict(perturbed)
        diff = np.mean(np.abs(baseline_pred - perturbed_pred))
        importance[idx] = diff
feature_dim = feature_array.shape[1]
if importance.size < feature_dim:
    padded = np.zeros(feature_dim)
    padded[: importance.size] = importance
    importance = padded
if importance.size:
    importance = importance / (np.sum(np.abs(importance)) + 1e-9)
shell_attention: List[float] = []
if importance.size:
    total_shells = max(1, feature_dim // shell_size)
    for shell_idx in range(total_shells):
        start = shell_idx * shell_size
        end = min(start + shell_size, importance.size)
        shell_attention.append(float(np.sum(importance[start:end])))
explanation = {
    "importance": importance.tolist(),
    "shellAttention": shell_attention,
```

```
            "method": method,
            "taskType": task_type,
        }
        return explanation


class BaselineModelSuite:
    """Trains classical ML baselines for comparison with quantum agents."""

    def __init__(
        self,
        validation_engine: StatisticalValidationEngine,
        explainability: ExplainabilityEngine,
        random_seed: int = DEFAULT_RANDOM_SEED,
    ) -> None:
        self.validation_engine = validation_engine
        self.explainability = explainability
        self.random_seed = random_seed
        self.results: Dict[str, Any] = {}

    def _build_models(self, task_type: str) -> List[Tuple[str, Callable[[], Any]]]:
        models: List[Tuple[str, Callable[[], Any]]] = []
        if task_type == "classification":
            if RandomForestClassifier is not None:
                models.append(("random_forest_classifier", lambda:
RandomForestClassifier(n_estimators=128, random_state=self.random_seed)))
            if LogisticRegression is not None:
                models.append(("logistic_regression", lambda: LogisticRegression(max_iter=200,
solver="lbfgs")))
            if TORCH_AVAILABLE:
                models.append(("graph_surrogate_classifier", lambda:
SimpleClassifier("baseline-graph")))
            else:
                models.append(("simple_classifier", lambda: SimpleClassifier("baseline-simple")))
        else:
            if RandomForestRegressor is not None:
                models.append(("random_forest_regressor", lambda:
RandomForestRegressor(n_estimators=128, random_state=self.random_seed)))
            if MLPRegressor is not None:
                models.append(("mlp_regressor", lambda: MLPRegressor(hidden_layer_sizes=(128,
64), random_state=self.random_seed, max_iter=300)))
            models.append(("graph_surrogate_regressor", lambda:
GraphSurrogateModel("baseline-graph")))
        return models
```

```
    def train_and_evaluate(self, dataset_manager: DatasetManager) -> Dict[str, Any]:
        for task, records in dataset_manager.records.items():
            split = dataset_manager.get_split(task) or dataset_manager.build_split(task)
            task_type = dataset_manager.get_task_type(task)
            models = self._build_models(task_type)
            evaluations: Dict[str, Dict[str, Any]] = {}
            explanations: Dict[str, Any] = {}
            for model_name, builder in models:
                model = builder()
                if hasattr(model, "fit"):
                    model.fit(split.train_X, split.train_y)
                elif isinstance(model, MLModelBase):
                    model.train(split)
                else:
                    continue
                if task_type == "classification":
                    if hasattr(model, "predict_proba"):
                        scores = model.predict_proba(split.test_X)[:, -1] if split.test_X.size else
model.predict_proba(split.train_X)[:, -1]
                    else:
                        scores = model.predict(split.test_X if split.test_X.size else split.train_X)
                    metrics = self.validation_engine.classification_metrics(split.test_y if split.test_y.size
else split.train_y, scores)
                else:
                    preds = model.predict(split.test_X) if split.test_X.size else
model.predict(split.train_X)
                    metrics = self.validation_engine.regression_metrics(split.test_y if split.test_y.size
else split.train_y, preds)
                evaluations[model_name] = metrics
                feature_sample = split.test_X if split.test_X.size else split.train_X
                explanation = self.explainability.explain(model, feature_sample if feature_sample.ndim
> 1 else feature_sample.reshape(-1, feature_sample.shape[0]), task_type)
                explanations[model_name] = explanation
            comparison = self.validation_engine.compare_model_variants(evaluations)
            self.results[task] = {
                "evaluations": evaluations,
                "comparison": comparison,
                "explanations": explanations,
                "taskType": task_type,
            }
        return self.results

class MLModelBase:
```

```
    """Abstract base class for ML models in the simulation."""

    def __init__(self, name: str, task_type: str, architecture: str) -> None:
        self.name = name
        self.task_type = task_type
        self.architecture = architecture
        self.version = 1
        self.metrics: Dict[str, Any] = {}

    def train(self, split: DatasetSplit) -> Dict[str, Any]:  # pragma: no cover - overridden
        raise NotImplementedError

    def predict(self, features: np.ndarray) -> np.ndarray:  # pragma: no cover - overridden
        raise NotImplementedError

    def predict_with_uncertainty(self, features: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        preds = self.predict(features)
        return preds, np.full_like(preds, 0.1, dtype=float)

    def describe(self) -> Dict[str, Any]:
        return {
            "name": self.name,
            "taskType": self.task_type,
            "architecture": self.architecture,
            "version": self.version,
            "metrics": self.metrics,
        }


class SimpleRegressor(MLModelBase):
    def __init__(self, name: str, architecture: str = "BayesianLinear") -> None:
        super().__init__(name, "regression", architecture)
        self.weights: Optional[np.ndarray] = None
        self.noise_variance: float = 0.1

    def train(self, split: DatasetSplit) -> Dict[str, Any]:
        X = split.train_X
        y = split.train_y
        if X.size == 0:
            self.weights = None
            self.noise_variance = 0.5
            self.metrics = {"mae": float("nan"), "rmse": float("nan")}
            return self.metrics
        X_aug = np.concatenate([X, np.ones((len(X), 1))], axis=1)
```

```
        try:
            self.weights = np.linalg.lstsq(X_aug, y, rcond=None)[0]
        except np.linalg.LinAlgError:
            self.weights = np.zeros(X_aug.shape[1])
        train_pred = self._predict_internal(X)
        residuals = y - train_pred
        self.noise_variance = float(np.maximum(np.var(residuals), 1e-3))
        val_pred = self.predict(split.val_X) if split.val_X.size else np.array([])
        mae = float(np.mean(np.abs(split.val_y - val_pred))) if val_pred.size else
float(np.mean(np.abs(residuals)))
        rmse = float(np.sqrt(np.mean((split.val_y - val_pred) ** 2))) if val_pred.size else
float(np.sqrt(np.mean(residuals ** 2)))
        self.metrics = {"mae": mae, "rmse": rmse}
        return self.metrics

    def _predict_internal(self, X: np.ndarray) -> np.ndarray:
        if self.weights is None:
            return np.zeros(X.shape[0])
        X_aug = np.concatenate([X, np.ones((len(X), 1))], axis=1)
        return X_aug @ self.weights

    def predict(self, features: np.ndarray) -> np.ndarray:
        if features.size == 0:
            return np.zeros(0)
        return self._predict_internal(features)

    def predict_with_uncertainty(self, features: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        preds = self.predict(features)
        if features.size == 0:
            return preds, np.zeros(0)
        variances = np.full(preds.shape, self.noise_variance + 0.05)
        return preds, np.sqrt(variances)


class SimpleClassifier(MLModelBase):
    def __init__(self, name: str, architecture: str = "LogisticRegressionLite") -> None:
        super().__init__(name, "classification", architecture)
        self.weights: Optional[np.ndarray] = None

    def train(self, split: DatasetSplit) -> Dict[str, Any]:
        X = split.train_X
        y = split.train_y
        if X.size == 0:
            self.weights = None
```

```
            self.metrics = {"accuracy": float("nan")}
            return self.metrics
        X_aug = np.concatenate([X, np.ones((len(X), 1))], axis=1)
        weights = np.zeros(X_aug.shape[1])
        lr = 0.1
        for _ in range(200):
            logits = X_aug @ weights
            probs = 1.0 / (1.0 + np.exp(-logits))
            gradient = X_aug.T @ (probs - y) / len(y)
            weights -= lr * gradient
        self.weights = weights
        val_pred = self.predict(split.val_X) if split.val_X.size else np.array([])
        if val_pred.size:
            accuracy = float(np.mean((val_pred > 0.5) == split.val_y))
        else:
            train_pred = self.predict(split.train_X)
            accuracy = float(np.mean((train_pred > 0.5) == y))
        self.metrics = {"accuracy": accuracy}
        return self.metrics

    def predict(self, features: np.ndarray) -> np.ndarray:
        if self.weights is None or features.size == 0:
            return np.zeros(features.shape[0] if features.ndim else 1)
        X_aug = np.concatenate([features, np.ones((len(features), 1))], axis=1)
        logits = X_aug @ self.weights
        return 1.0 / (1.0 + np.exp(-logits))

    def predict_with_uncertainty(self, features: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        preds = self.predict(features)
        uncertainty = np.sqrt(preds * (1.0 - preds) + 1e-3)
        return preds, uncertainty


class GraphSurrogateModel(SimpleRegressor):
    """Message-passing inspired surrogate operating on aggregated graph features."""

    def __init__(self, name: str) -> None:
        super().__init__(name, architecture="MessagePassingSurrogate")


class MLModelRegistry:
    """Tracks model instances, metrics, and versioned training provenance."""

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

```
        self.models: Dict[str, MLModelBase] = {}
        self.training_log: List[Dict[str, Any]] = []

    def register_model(self, task: str, model: MLModelBase, metrics: Dict[str, Any], dataset_meta:
Dict[str, Any]) -> None:
        model.version = self.models.get(task, model).version + (1 if task in self.models else 0)
        self.models[task] = model
        self.training_log.append({
            "task": task,
            "version": model.version,
            "metrics": metrics,
            "dataset": dataset_meta,
        })

    def get_model(self, task: str) -> Optional[MLModelBase]:
        return self.models.get(task)

    def describe(self) -> Dict[str, Any]:
        return {
            "trainedModels": [model.describe() for model in self.models.values()],
            "trainingLog": list(self.training_log),
        }


class ActiveLearningCoordinator:
    """Implements active learning with uncertainty-driven sampling."""

    def __init__(self, threshold: float = 0.2) -> None:
        self.threshold = threshold
        self.pending_samples: List[Dict[str, Any]] = []
        self.completed_jobs: List[Dict[str, Any]] = []

    def evaluate_samples(
        self,
        task: str,
        features: Iterable[np.ndarray],
        predictions: Iterable[float],
        uncertainties: Iterable[float],
        metadata: Iterable[Dict[str, Any]],
    ) -> None:
        for feat, pred, unc, meta in zip(features, predictions, uncertainties, metadata):
            if float(unc) >= self.threshold:
                self.pending_samples.append({
                    "task": task,
```

```
                "prediction": float(pred),
                "uncertainty": float(unc),
                "metadata": meta,
                "featurePreview": np.asarray(feat, dtype=float)[:5].tolist(),
            })

    def schedule_retraining(self, task: str, registry: MLModelRegistry, dataset: DatasetManager)
-> None:
        if not self.pending_samples:
            return
        job = {
            "task": task,
            "pending": len(self.pending_samples),
            "registrySize": len(registry.models),
            "datasetRecords": {key: len(val) for key, val in dataset.records.items()},
        }
        self.completed_jobs.append(job)
        self.pending_samples.clear()

    def describe(self) -> Dict[str, Any]:
        return {
            "threshold": self.threshold,
            "pending": list(self.pending_samples),
            "completed": list(self.completed_jobs),
        }


class MLInferenceAPI:
    """Lightweight registry for auditable ML inference endpoints."""

    def __init__(self) -> None:
        self.endpoints: Dict[str, Dict[str, Any]] = {}
        self.logs: deque = deque(maxlen=50)

    def register_endpoint(self, name: str, task: str, model_version: int) -> None:
        self.endpoints[name] = {
            "task": task,
            "modelVersion": model_version,
        }

    def log_call(self, endpoint: str, payload: Dict[str, Any], response: Dict[str, Any]) -> None:
        self.logs.appendleft({
            "endpoint": endpoint,
            "payload": payload,
```

```
            "response": response,
        })

    def describe(self) -> Dict[str, Any]:
        return {
            "endpoints": dict(self.endpoints),
            "recentCalls": list(self.logs),
        }


# ---------------------------------------------------------------------------
# Public data clients (with offline fallbacks)
# ---------------------------------------------------------------------------


class PublicDataClient:
    def __init__(self) -> None:
        pass

    def _http_get(self, url: str) -> Optional[bytes]:
        try:
            with urlrequest.urlopen(url, timeout=10) as response:  # nosec B310
                return response.read()
        except (urlerror.URLError, TimeoutError):
            return None

    def _http_post(self, url: str, payload: Dict[str, Any]) -> Optional[bytes]:
        try:
            data = json.dumps(payload).encode("utf-8")
            req = urlrequest.Request(url, data=data, headers={"Content-Type": "application/json"})
            with urlrequest.urlopen(req, timeout=10) as response:  # nosec B310
                return response.read()
        except (urlerror.URLError, TimeoutError):
            return None

    def fetch_pdb(self, pdb_id: str) -> str:
        url = f"https://files.rcsb.org/download/{pdb_id}.pdb"
        content = self._http_get(url)
        if content:
            try:
                return content.decode("utf-8")
            except UnicodeDecodeError:
                return content.decode("latin-1", errors="ignore")
        # Fallback: alpha helix snippet
```

```
        return textwrap.dedent(
            """
            ATOM      1  N   MET A   1      11.104  13.207   9.100  1.00 20.00           N
            ATOM      2  CA  MET A   1      12.560  13.320   9.300  1.00 20.00           C
            ATOM      3  C   MET A   1      13.080  14.740   8.900  1.00 20.00           C
            ATOM      4  O   MET A   1      12.540  15.780   9.300  1.00 20.00           O
            ATOM      5  CB  MET A   1      13.220  12.200  10.200  1.00 20.00           C
            ATOM      6  N   ALA A   2      14.180  14.860   8.100  1.00 20.00           N
            ATOM      7  CA  ALA A   2      14.820  16.170   7.700  1.00 20.00           C
            ATOM      8  C   ALA A   2      16.330  16.120   8.100  1.00 20.00           C
            ATOM      9  O   ALA A   2      17.090  15.170   7.800  1.00 20.00           O
            ATOM     10  CB  ALA A   2      14.600  16.430   6.200  1.00 20.00           C
            TER
            END
            """
        ).strip()

    def fetch_pubchem_candidates(self, query: str) -> List[Dict[str, Any]]:
        url = (
            "https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/name/"
f"{query}/property/CanonicalSMILES,MolecularWeight,HBondDonorCount,HBondAcceptorCount
/JSON"
        )
        content = self._http_get(url)
        if content:
            try:
                data = json.loads(content.decode("utf-8"))
                props = data.get("PropertyTable", {}).get("Properties", [])
                results = []
                for idx, prop in enumerate(props):
                    results.append(
                        {
                            "ligandId": f"pubchem-{query}-{idx}",
                            "smiles": prop.get("CanonicalSMILES", ""),
                            "molecularWeight": prop.get("MolecularWeight"),
                            "donors": prop.get("HBondDonorCount"),
                            "acceptors": prop.get("HBondAcceptorCount"),
                        }
                    )
                if results:
                    return results
            except json.JSONDecodeError:
                pass
```

```
        return [
            {
                "ligandId": "fallback-aspirin",
                "smiles": "CC(=O)OC1=CC=CC=C1C(=O)O",
                "molecularWeight": 180.16,
                "donors": 1,
                "acceptors": 4,
            }
        ]

    def fetch_uniprot_metadata(self, accession: str) -> Dict[str, Any]:
        url = f"https://rest.uniprot.org/uniprotkb/{accession}.json"
        content = self._http_get(url)
        if content:
            try:
                data = json.loads(content.decode("utf-8"))
                protein = data.get("proteinDescription", {}).get("recommendedName",
{}).get("fullName", {}).get("value")
                organism = data.get("organism", {}).get("scientificName")
                return {"protein": protein, "organism": organism}
            except json.JSONDecodeError:
                pass
        return {"protein": "Cyclooxygenase-2", "organism": "Homo sapiens"}

    def fetch_patent_hits(self, query: str) -> List[Dict[str, Any]]:
        url = "https://api.patentsview.org/patents/query"
        payload = {
            "q": {"_text_any": {"patent_title": query}},
            "f": ["patent_number", "patent_title"],
            "o": {"per_page": 5},
        }
        content = self._http_post(url, payload)
        if content:
            try:
                data = json.loads(content.decode("utf-8"))
                patents = data.get("patents", [])
                return [
                    {
                        "patent": entry.get("patent_number"),
                        "title": entry.get("patent_title"),
                    }
                    for entry in patents
                ]
            except json.JSONDecodeError:
```

```
            pass
        return [
            {"patent": "US-12345-B2", "title": "Aspirin formulations with enhanced stability"},
            {"patent": "US-98765-C1", "title": "Novel COX-2 inhibitors"},
        ]


# ----------------------------------------------------------------------
# Quantum analytics helpers
# ----------------------------------------------------------------------


class LambdaScalingToolkit:
    """Utility collection for λ-shell embeddings and diagnostics."""

    DEFAULT_SHELL_COUNT = 5

    @staticmethod
    def _bhattacharyya(p: float, q: float) -> float:
        p = float(np.clip(p, 1e-6, 1.0))
        q = float(np.clip(q, 1e-6, 1.0))
        return float(np.clip(math.sqrt(p * q), 0.0, 1.0))

    @classmethod
    def _as_descriptor(
        cls,
        index: int,
        radius: float,
        curvature: float,
        entropy: float,
        energy_density: float,
        occupancy: float,
        expected: float,
    ) -> LambdaShellDescriptor:
        overlap = cls._bhattacharyya(occupancy, expected)
        leakage = float(np.clip(abs(occupancy - expected), 0.0, 1.0))
        return LambdaShellDescriptor(
            shell_index=index,
            lambdaRadius=float(np.clip(radius, 1e-3, 1e3)),
            lambdaCurvature=float(np.clip(curvature, -200.0, 200.0)),
            lambdaEntropy=float(np.clip(entropy, 0.0, 10.0)),
            lambdaEnergyDensity=float(np.clip(energy_density, -500.0, 500.0)),
            lambdaBhattacharyya=overlap,
            lambdaOccupancy=float(np.clip(occupancy, 0.0, 2.0)),
```

```
            lambdaLeakage=leakage,
        )

    @classmethod
    def compute_base_descriptors(
        cls,
        curvature_profile: Iterable[float],
        lambda_modes: Iterable[float],
        entanglement_entropy: float,
        radii: Iterable[float],
        shells: Iterable[int],
        energy_stats: Dict[str, Any],
    ) -> Tuple[List[Dict[str, Any]], Dict[str, Any]]:
        curvature_list = list(curvature_profile) or [0.0]
        mode_list = list(lambda_modes) or [1.0]
        radii_array = np.asarray(list(radii) or [1.0], dtype=float)
        shells_array = np.asarray(list(shells) or [0], dtype=int)
        unique_shells = sorted(set(int(val) for val in shells_array.tolist()))
        if not unique_shells:
            unique_shells = list(range(cls.DEFAULT_SHELL_COUNT))
        shell_count = min(len(unique_shells), cls.DEFAULT_SHELL_COUNT)
        base_radius = float(np.max(np.abs(radii_array)) or 1.0)
        descriptors: List[LambdaShellDescriptor] = []
        total_points = max(int(len(radii_array)), 1)
        expected = 1.0 / float(shell_count)
        for idx in range(shell_count):
            shell_id = unique_shells[idx]
            mask = shells_array == shell_id
            occupancy = float(np.count_nonzero(mask) / total_points)
            radius = base_radius / math.pow(LAMBDA_DILATION, idx)
            curvature = curvature_list[idx % len(curvature_list)]
            entropy = float(
                np.clip(
                    entanglement_entropy * (1.0 + 0.05 * idx),
                    0.001,
                    10.0,
                )
            )
            energy_mode = mode_list[idx % len(mode_list)]
            energy_density = float(energy_mode / max(radius, 1e-3))
            descriptors.append(
                cls._as_descriptor(idx, radius, curvature, entropy, energy_density, occupancy,
expected)
            )
```

```
        attractor_score = float(np.mean([d.lambdaBhattacharyya for d in descriptors]))
        entropy_gradient = float(descriptors[-1].lambdaEntropy - descriptors[0].lambdaEntropy) if
descriptors else 0.0
        bhattacharyya_flux = float(np.std([d.lambdaBhattacharyya for d in descriptors])) if
descriptors else 0.0
        basis = {
            "dilationFactor": LAMBDA_DILATION,
            "shellCount": len(descriptors),
            "lambdaAttractorScore": float(np.clip(attractor_score, 0.0, 1.0)),
            "lambdaEntropyGradient": float(np.clip(entropy_gradient, -5.0, 5.0)),
            "lambdaBhattacharyyaFlux": float(np.clip(bhattacharyya_flux, 0.0, 1.0)),
            "referenceEnergyMean": energy_stats.get("meanEnergy"),
        }
        return [descriptor.to_dict() for descriptor in descriptors], basis

    @staticmethod
    def fingerprint(descriptors: Iterable[Dict[str, Any]]) -> List[float]:
        vector: List[float] = []
        for entry in descriptors:
            vector.extend(
                [
                    float(entry.get("lambdaRadius", 0.0)),
                    float(entry.get("lambdaCurvature", 0.0)),
                    float(entry.get("lambdaEntropy", 0.0)),
                    float(entry.get("lambdaEnergyDensity", 0.0)),
                    float(entry.get("lambdaBhattacharyya", 0.0)),
                    float(entry.get("lambdaOccupancy", 0.0)),
                    float(entry.get("lambdaLeakage", 0.0)),
                ]
            )
        if not vector:
            vector = [0.0] * 7
        return vector

    @classmethod
    def analyze_coordinates(cls, coords: np.ndarray, context: QuantumContext) -> Dict[str, Any]:
        base_descriptors = context.lambda_shells or []
        if not len(base_descriptors):
            return {"descriptors": [], "fingerprint": [0.0] * 7, "summary": {}}
        center = np.mean(coords, axis=0)
        distances = np.linalg.norm(coords - center, axis=1)
        max_radius = float(np.max(distances) or 1.0)
        thresholds = [max_radius / math.pow(LAMBDA_DILATION, idx) for idx, _ in
enumerate(base_descriptors)]
```

```
        descriptors: List[Dict[str, Any]] = []
        for idx, base in enumerate(base_descriptors):
            upper = thresholds[idx]
            lower = thresholds[idx + 1] if idx + 1 < len(thresholds) else 0.0
            mask = (distances <= upper) & (distances >= lower)
            occupancy = float(np.mean(mask)) if distances.size else 0.0
            expected = base.get("lambdaOccupancy", 1.0 / len(base_descriptors))
            curvature = base.get("lambdaCurvature", 0.0) * (1.0 + 0.05 * occupancy)
            entropy = base.get("lambdaEntropy", context.entanglement_entropy) * (1.0 + 0.02 * idx)
            energy_density = base.get("lambdaEnergyDensity", 0.0) * (1.0 + 0.03 * (occupancy -
expected))
            descriptor = cls._as_descriptor(
                idx,
                radius=upper or max_radius,
                curvature=curvature,
                entropy=entropy,
                energy_density=energy_density,
                occupancy=occupancy,
                expected=expected,
            ).to_dict()
            descriptor["lambdaBhattacharyya"] = cls._bhattacharyya(occupancy, expected)
            descriptor["lambdaLeakage"] = float(np.clip(abs(occupancy - expected), 0.0, 1.0))
            descriptors.append(descriptor)
        fingerprint = cls.fingerprint(descriptors)
        summary = {
            "lambdaShellFingerprint": fingerprint,
            "lambdaBhattacharyyaFlux": float(np.std([d["lambdaBhattacharyya"] for d in descriptors])
if descriptors else 0.0),
            "lambdaEntropyGradient": float(
                (descriptors[-1]["lambdaEntropy"] - descriptors[0]["lambdaEntropy"]) if len(descriptors)
> 1 else 0.0
            ),
            "scalingBasis": "lambda",
        }
        return {"descriptors": descriptors, "fingerprint": fingerprint, "summary": summary}

    @classmethod
    def analyze_ligand(
        cls,
        smiles: str,
        context: QuantumContext,
        quantum_reference: Dict[str, Any],
    ) -> Dict[str, Any]:
        base_descriptors = context.lambda_shells or []
```

```
        if not base_descriptors:
            return {"descriptors": [], "fingerprint": [0.0] * 7, "summary": {}}
        atom_counts = {atom: smiles.count(atom) for atom in ["C", "N", "O", "S", "P", "F", "Cl"]}
        total_atoms = float(sum(atom_counts.values()) or 1.0)
        descriptors: List[Dict[str, Any]] = []
        for idx, base in enumerate(base_descriptors):
            atom = ["C", "N", "O", "S", "P", "F", "Cl"][idx % 7]
            bias = atom_counts.get(atom, 0.0) / total_atoms
            expected = base.get("lambdaOccupancy", 1.0 / len(base_descriptors))
            occupancy = float(np.clip(expected * (1.0 + 0.5 * bias), 0.0, 2.0))
            entropy = base.get("lambdaEntropy", context.entanglement_entropy) * (1.0 + 0.1 * bias)
            energy_density = base.get("lambdaEnergyDensity", 0.0) * (1.0 + 0.05 * bias)
            descriptor = cls._as_descriptor(
                idx,
                radius=base.get("lambdaRadius", 1.0),
                curvature=base.get("lambdaCurvature", 0.0) * (1.0 + 0.05 * (bias - 0.2)),
                entropy=entropy,
                energy_density=energy_density,
                occupancy=occupancy,
                expected=expected,
            ).to_dict()
            descriptors.append(descriptor)
        fingerprint = cls.fingerprint(descriptors)
        mean_energy = quantum_reference.get("statistics", {}).get("meanEnergy", -8.0)
        summary = {
            "lambdaShellFingerprint": fingerprint,
            "lambdaMeanEnergyAlignment": float(
                np.clip(1.0 - abs(mean_energy) / (abs(mean_energy) + 10.0), 0.0, 1.0)
            ),
            "lambdaOccupancyMean": float(np.mean([d["lambdaOccupancy"] for d in descriptors]) if
descriptors else 0.0),
            "scalingBasis": "lambda",
        }
        return {"descriptors": descriptors, "fingerprint": fingerprint, "summary": summary}


class QuantumMemoryAPI:
    """Compresses and restores lambda/phi scaled memory vectors."""

    def __init__(self) -> None:
        self.alpha = DUAL_SCALING_ALPHA
        self.storage: List[Dict[str, Any]] = []

    @staticmethod
```

```
    def _convert_basis(vector: np.ndarray, source: str, target: str, alpha: float) -> np.ndarray:
        if source == target:
            return vector
        safe = np.sign(vector) * np.power(np.abs(vector) + 1e-9, alpha if source == "lambda" else
1.0 / alpha)
        return safe

    @staticmethod
    def _geometric_params(vector: np.ndarray) -> Optional[Dict[str, Any]]:
        if vector.size < 3:
            return None
        non_zero = np.where(np.abs(vector) > 1e-8)[0]
        if non_zero.size < 2:
            return None
        ratios = []
        for idx in range(len(vector) - 1):
            if abs(vector[idx]) < 1e-8:
                continue
            ratios.append(vector[idx + 1] / vector[idx])
        if not ratios:
            return None
        ratio_mean = float(np.mean(ratios))
        if abs(ratio_mean - LAMBDA_DILATION) < 0.05:
            basis = "lambda"
        elif abs(ratio_mean - PHI_CONSTANT) < 0.05:
            basis = "phi"
        else:
            return None
        return {
            "basis": basis,
            "seed": float(vector[0]),
            "ratio": ratio_mean,
            "length": int(vector.size),
        }

    def serialize_memory(
        self,
        state_vector: Iterable[float],
        metadata: Optional[Dict[str, Any]] = None,
        canonical_basis: str = "phi",
    ) -> Dict[str, Any]:
        metadata = copy.deepcopy(metadata) if metadata else {}
        basis = metadata.get("scaling_basis", "lambda")
        array = np.asarray(list(state_vector), dtype=float)
```

```
        compression = self._geometric_params(array)
        entry: Dict[str, Any] = {
            "metadata": metadata,
            "basis": basis,
            "canonical_basis": canonical_basis,
        }
        if compression:
            entry["compressed"] = compression
            entry["metadata"]["compression"] = "geometric"
        else:
            entry["raw"] = array.tolist()
            entry["metadata"]["compression"] = "raw"
        entry["metadata"]["scaling_basis"] = basis
        entry["metadata"]["canonical_basis"] = canonical_basis
        self.storage.append(entry)
        return entry

    def deserialize_memory(self, entry: Dict[str, Any]) -> np.ndarray:
        basis = entry.get("basis", "lambda")
        canonical = entry.get("canonical_basis", "phi")
        if "compressed" in entry:
            comp = entry["compressed"]
            values = [comp["seed"] * (comp["ratio"] ** idx) for idx in range(comp.get("length", 0))]
            vector = np.asarray(values, dtype=float)
        else:
            vector = np.asarray(entry.get("raw", []), dtype=float)
        converted = self._convert_basis(vector, basis, canonical, self.alpha)
        return converted


class QuantumPhysicsEngine:
    def __init__(self, geometry: GeometryParams, field: FieldParams, ent_params:
EntanglementParams):
        self.geometry = geometry
        self.field = field
        self.ent_params = ent_params

    def compute_quantum_context(self) -> QuantumContext:
        z, r, rho, curvature = integrate_profile(self.geometry)
        operator, _ = build_kg_operator(z, r, curvature, self.field)
        eigenvalues, _ = compute_modes(operator, k=min(5, len(z) - 2))
        curvature_profile = curvature.tolist()
        lambda_modes = [float(val) for val in eigenvalues]
```

```
        positions, radii, shells = build_geometry(self.ent_params)
        adjacency = build_adjacency(positions, radii, shells, self.ent_params)
        hamiltonian = build_hamiltonian(adjacency, radii, self.ent_params)
        # compute ground state entropy across first three shells
        evals, evecs = np.linalg.eigh(hamiltonian.toarray())
        ground_state = evecs[:, np.argmin(evals)]
        mask = shells < 3
        entropy = single_particle_entropy_for_cut(ground_state, mask)

        enhancement = float(np.clip(np.mean(curvature) * 1e-2 + statistics.mean(lambda_modes),
0.5, 5.0))
        lambda_shells, lambda_basis = LambdaScalingToolkit.compute_base_descriptors(
            curvature_profile,
            lambda_modes,
            float(entropy),
            radii,
            shells,
            {"meanEnergy": float(np.mean(lambda_modes)) if lambda_modes else None},
        )
        return QuantumContext(
            curvature_profile=curvature_profile,
            lambda_modes=lambda_modes,
            entanglement_entropy=float(entropy),
            enhancement_factor=enhancement,
            lambda_shells=lambda_shells,
            lambda_basis=lambda_basis,
        )


class QuantumCircuitEngine:
    """Generates quantum-consistent exemplars via lightweight circuit sampling."""

    def __init__(self, context: QuantumContext, seed: int = 314159) -> None:
        self.context = context
        self.rng = np.random.default_rng(seed)

    def _simulate_circuit(self, ligand_id: str) -> Dict[str, Any]:
        base_energy = -12.0 - 0.3 * self.context.enhancement_factor
        noise = self.rng.normal(0, 0.6)
        binding_energy = float(np.clip(base_energy + noise, -60.0, -0.5))
        entropy = float(
            np.clip(
                self.context.entanglement_entropy + self.rng.normal(0, 0.05),
                0.01,
```

```
                1.5 * max(0.1, self.context.entanglement_entropy),
            )
        )
        occupation = self.rng.dirichlet(np.ones(4))
        transition_probs = occupation.tolist()
        fidelity = float(np.clip(0.98 + self.rng.normal(0, 0.005), 0.85, 0.999))
        shell_fp = LambdaScalingToolkit.fingerprint(self.context.lambda_shells)
        return {
            "ligandId": ligand_id,
            "bindingEnergy": binding_energy,
            "entanglementEntropy": entropy,
            "orbitalOccupations": transition_probs,
            "fidelity": fidelity,
            "lambdaShellFingerprint": shell_fp,
        }

    def generate_reference_dataset(self, ligand_ids: List[str]) -> Dict[str, Any]:
        samples = [self._simulate_circuit(ligand_id) for ligand_id in ligand_ids]
        energies = [entry["bindingEnergy"] for entry in samples]
        entropies = [entry["entanglementEntropy"] for entry in samples]
        stats = {
            "energyRange": {"min": float(min(energies)), "max": float(max(energies))},
            "entropyRange": {"min": float(min(entropies)), "max": float(max(entropies))},
            "meanEnergy": float(np.mean(energies)),
            "meanEntropy": float(np.mean(entropies)),
        }
        self.context.lambda_basis.update(
            {
                "referenceEnergyMean": stats["meanEnergy"],
                "lambdaEntropyGradient": float(stats["meanEntropy"] -
self.context.entanglement_entropy),
            }
        )
        return {"samples": samples, "statistics": stats}


class PhysicalValidator:
    """Applies physical constraints, uncertainty propagation, and rationale tracing."""

    def __init__(self, context: QuantumContext, reference_stats: Dict[str, Any]):
        self.context = context
        self.reference_stats = reference_stats
        self.rng = np.random.default_rng(2718)
        self.numeric_constraints = {
```

```
        "druggabilityScore": (0.0, 1.0),
        "hydrophobicity": (0.0, 1.0),
        "electrostaticPotential": (-20.0, 0.0),
        "bindingAffinityScore": (-60.0, -0.5),
        "bindingFreeEnergy": (-80.0, -0.5),
        "shapeComplementarity": (0.0, 1.0),
        "toxicityRiskScore": (0.0, 1.0),
        "bioavailability": (0.0, 1.0),
        "halfLife": (0.1, 240.0),
        "syntheticAccessibility": (1.0, 10.0),
        "noveltyScore": (0.0, 1.0),
        "confidence": (0.0, 1.0),
        "quantumCircuitFidelity": (0.0, 1.0),
        "lambdaEnhancement": (0.0, 10.0),
        "bindingEnergyRef": (-80.0, -0.5),
        "bindingEnergy": (-80.0, -0.5),
        "entanglementEntropyRef": (0.0, 5.0),
        "entanglementEntropy": (0.0, 5.0),
        "matchingScore": (0.0, 1.0),
        "meanEntanglementEntropy": (0.0, 5.0),
        "entropyStdDev": (0.0, 5.0),
        "canonicalBeta": (0.0, 200.0),
        "partitionFunction": (0.0, 1e4),
        "lambdaRadius": (0.0, 1e3),
        "lambdaCurvature": (-500.0, 500.0),
        "lambdaEntropy": (0.0, 10.0),
        "lambdaEnergyDensity": (-500.0, 500.0),
        "lambdaBhattacharyya": (0.0, 1.0),
        "lambdaOccupancy": (0.0, 2.0),
        "lambdaLeakage": (0.0, 1.0),
        "lambdaAttractorScore": (0.0, 1.0),
        "lambdaEntropyGradient": (-5.0, 5.0),
        "lambdaBhattacharyyaFlux": (0.0, 1.0),
    }

def _estimate_sigma(self, key: str, value: float) -> float:
    baseline = 0.05 * max(1.0, abs(value))
    ent_scale = 0.02 * max(1.0, self.context.entanglement_entropy)
    ref = self.reference_stats.get("statistics", {})
    if "meanEnergy" in ref and key in {"bindingAffinityScore", "bindingFreeEnergy"}:
        baseline += 0.1 * abs(value - ref["meanEnergy"])
    return float(np.clip(baseline + ent_scale, 0.01, 5.0))

def _credible_interval(self, mean: float, sigma: float) -> Tuple[float, float]:
```

```
    samples = self.rng.normal(mean, sigma, size=2000)
    lower, upper = np.percentile(samples, [5, 95])
    return float(lower), float(upper)

def _coerce_value(self, key: str, value: float) -> Tuple[float, bool]:
    constraint = self.numeric_constraints.get(key)
    adjusted = False
    if constraint:
        min_val, max_val = constraint
        if value < min_val:
            value = min_val
            adjusted = True
        if value > max_val:
            value = max_val
            adjusted = True
    if key in {"bindingAffinityScore", "bindingFreeEnergy"}:
        ref = self.reference_stats.get("statistics", {})
        energy_range = ref.get("energyRange")
        if energy_range:
            if value < energy_range["min"]:
                value = float(energy_range["min"])
                adjusted = True
            if value > energy_range["max"]:
                value = float(energy_range["max"])
                adjusted = True
    return float(value), adjusted

def validate(self, agent: str, payload: Dict[str, Any]) -> Dict[str, Any]:
    metadata: Dict[str, Any] = {"agent": agent, "adjustments": []}

    root_holder: Dict[str, Any] = {"root": copy.deepcopy(payload)}

    def _locate(container: Any, key: str) -> Any:
        if key.startswith("["):
            index = int(key.strip("[]"))
            return container[index]
        return container[key]

    def _process(obj: Any, path: Tuple[str, ...]) -> Any:
        if isinstance(obj, dict):
            for key, value in list(obj.items()):
                new_path = path + (key,)
                obj[key] = _process(value, new_path)
            return obj
```

```
            if isinstance(obj, list):
                for idx, item in enumerate(obj):
                    obj[idx] = _process(item, path + (f"[{idx}]",))
                return obj
            if isinstance(obj, (int, float)):
                corrected, adjusted = self._coerce_value(path[-1] if path else "", float(obj))
                sigma = self._estimate_sigma(path[-1] if path else "", corrected)
                lower, upper = self._credible_interval(corrected, sigma)
                container: Any = root_holder["root"]
                for key in path[:-1]:
                    container = _locate(container, key)
                if isinstance(container, dict) and path:
                    container[f"{path[-1]}Uncertainty"] = {
                        "stdDev": sigma,
                        "credibleInterval": [lower, upper],
                    }
                if adjusted:
                    metadata["adjustments"].append({"path": ".".join(path), "reason": "Constraint
enforcement"})
                return corrected
            return obj

        root_holder["root"] = _process(root_holder["root"], tuple())
        validated_payload = root_holder["root"]
        validated_payload.setdefault("validation", {}).update({
            "agent": agent,
            "adjustmentCount": len(metadata["adjustments"]),
            "referenceEnergyRange": self.reference_stats.get("statistics", {}).get("energyRange"),
        })
        if metadata["adjustments"]:
            validated_payload["validation"]["adjustments"] = metadata["adjustments"]
        return validated_payload



# ---------------------------------------------------------------------
# Golden Turing AI integration adapter
# ---------------------------------------------------------------------


class GoldenTuringDDSAdapter:
    """Map Golden Turing AI recursive features onto the DDS pipeline."""

    def __init__(
```

```
    self,
    core_ai: Any,
    blackboard: QuantumBlackboard,
    validator: PhysicalValidator,
    quantum_reference: Dict[str, Any],
    memory_api: Optional[QuantumMemoryAPI] = None,
) -> None:
    self.core_ai = core_ai
    self.blackboard = blackboard
    self.validator = validator
    self.quantum_reference = quantum_reference
    self.memory_api = memory_api or QuantumMemoryAPI()
    self.state_potential = 0.62
    self.self_awareness = 0.72
    self.state_resonance = 0.28
    self.potential_awareness_boost_factor = 0.25
    self.entanglement_param_boost_factor = 1.5
    self.tunneling_state_shift_factor = 0.3
    self.memory_fidelity_noise_factor = 0.005
    self.simulation_param_variation_scale = 0.1
    self.simulation_multiverse_params = 3
    self.annealing_stability_threshold = 0.01
    self.annealing_lr_factor_stable = 0.9
    self.annealing_lr_factor_unstable = 1.1
    self.annealing_mutation_scale_factor_stable = 0.5
    self.annealing_mutation_scale_factor_unstable = 1.2
    self.interference_blend_factor_base = 0.05
    self.zeno_effect_trigger_count = 5
    self.zeno_effect_time_window = 60.0
    self.zeno_effect_dampening_factor = 0.1
    self.awareness_history: deque = deque([self.self_awareness], maxlen=200)
    self.awareness_changes: List[float] = []
    self.agent_roles: Dict[str, str] = {}
    self.action_log: List[Dict[str, Any]] = []
    self.pending_blackboard_posts: List[Dict[str, Any]] = []
    self.entanglement_records: List[Dict[str, Any]] = []
    self.annealing_records: List[Dict[str, Any]] = []
    self.tunneling_records: List[Dict[str, Any]] = []
    self.blend_records: List[Dict[str, Any]] = []
    self.simulation_reflections: List[Dict[str, Any]] = []
    self.meta_tuning_records: List[Dict[str, Any]] = []
    self.memory_noise_records: List[Dict[str, Any]] = []
    self.memory_cache: Dict[str, List[Dict[str, Any]]] = {"self": [], "adversary": []}
    self.agent_map: Dict[str, AgentBase] = {}
```

```
    self.binding_history: deque = deque(maxlen=60)
    self.pains_alert_counter = 0
    self.stagnation_counter = 0
    self.current_target_focus = "pocket-01"
    self.analysis_action_window: deque = deque()
    self.zeno_dampening_active = False

def register_agent(self, agent: Any) -> None:
    self.agent_roles[agent.name] = getattr(agent.__class__, "__name__", agent.name)
    self.agent_map[agent.name] = agent

def before_agent_run(self, agent_name: str) -> None:
    self._register_action_event("ACTION_ANALYZE_STATE", agent_name)

def after_agent_report(
    self,
    agent_name: str,
    report: Dict[str, Any],
) -> Tuple[Dict[str, Any], Dict[str, Any]]:
    integration_meta: Dict[str, Any] = {}
    awareness_delta = self._estimate_awareness_change(agent_name, report)
    if agent_name in {"LigandDiscoveryAgent", "QuantumSimulationAgent"}:
        self._record_binding_score(agent_name, report)
    if agent_name == "SafetyAgent":
        self._track_pains_alerts(report)
    superposition_meta = self._apply_superposition(agent_name, report)
    if superposition_meta:
        integration_meta["superposition"] = superposition_meta
    ent_meta = self._apply_entanglement_feedback(agent_name, awareness_delta)
    if ent_meta:
        integration_meta["entanglement"] = ent_meta
    awareness_change = self._update_awareness(awareness_delta)
    integration_meta["awarenessDelta"] = awareness_change
    annealing_meta = self._apply_adaptive_annealing()
    if annealing_meta:
        integration_meta["annealing"] = annealing_meta
    tunneling_meta = self._check_tunneling_conditions()
    if tunneling_meta:
        integration_meta["quantumTunneling"] = tunneling_meta
    if agent_name == "QuantumSimulationAgent":
        blend_meta = self._perform_state_blend()
        if blend_meta:
            integration_meta["interferenceBlend"] = blend_meta
    memory_meta = self._apply_memory_fidelity_noise(agent_name, report)
```

```
    if memory_meta:
        integration_meta["memoryFidelity"] = memory_meta
    self._register_memory_snapshot(agent_name, report)
    integration_meta.setdefault("lambdaScaling", {})
    integration_meta["lambdaScaling"].update(
        {
            "basis": self.validator.context.lambda_basis,
            "shellFingerprint": LambdaScalingToolkit.fingerprint(
                self.validator.context.lambda_shells
            )[:7],
        }
    )
    return report, integration_meta

def run_recursive_simulation(self) -> Dict[str, Any]:
    variations: List[Dict[str, Any]] = []
    base_reward = float(np.mean(self.binding_history)) if self.binding_history else -8.0
    for idx in range(self.simulation_multiverse_params):
        perturb = (idx - 1) * self.simulation_param_variation_scale
        success_rate = float(
            np.clip(
                0.55
                + 0.1 * random.random()
                + 0.05 * (-base_reward / 10.0)
                + perturb,
                0.0,
                1.0,
            )
        )
        variations.append(
            {
                "paramSetId": f"mv-{idx}",
                "ligandWeight": float(np.clip(0.6 + perturb, 0.1, 0.95)),
                "toxicityWeight": float(np.clip(0.3 - perturb, 0.05, 0.9)),
                "simulatedSuccessRate": success_rate,
            }
        )
    best = max(variations, key=lambda entry: entry["simulatedSuccessRate"])
    prompt = {
        "action": "RUN_SIMULATION",
        "payload": {
            "target_protein": self.current_target_focus,
            "iterations": 50,
            "levels": 5,
```

```
            "multiverse_count": self.simulation_multiverse_params,
            "strategy_focus": ["ligand_generation", "quantum_docking"],
            "param_variation_scale": self.simulation_param_variation_scale,
        },
    }
    self._queue_action("ACTION_RUN_SIMULATION", prompt)
    reflection = {
        "prompt": prompt,
        "variations": variations,
        "bestParamSetId": best["paramSetId"],
        "averageSuccessRate": float(
            np.mean([entry["simulatedSuccessRate"] for entry in variations])
        ),
        "awarenessBonus": 0.3,
    }
    self.simulation_reflections.append(reflection)
    self._update_awareness(0.03)
    return reflection

def tune_analysis_parameters(self, avg_reward: float) -> Dict[str, Any]:
    tuning_payload = {
        "action": "TUNE_DDS_METRICS",
        "payload": {
            "tuning_target": "agent_thresholds",
            "recent_avg_reward": avg_reward,
            "tuning_rate": 0.12,
            "noise_factor": 0.02,
        },
    }
    self._queue_action("ACTION_TUNE_ANALYSIS", tuning_payload)
    meta = {
        "analysisWeights": {
            "prediction_error_impact": float(np.clip(1.0 + avg_reward, 0.5, 1.5)),
            "timing_penalty_factor": float(np.clip(0.8 - avg_reward * 0.1, 0.2, 1.2)),
        },
        "ruleThresholds": {
            "simulation_trigger_awareness": float(np.clip(self.self_awareness - 0.05, 0.1, 0.9)),
        },
        "prompt": tuning_payload,
    }
    self.meta_tuning_records.append(meta)
    return meta

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

```
    low_awareness_flag = self.self_awareness < 0.5
    payload = {
        "action": "ANALYZE_MEMORY_FIDELITY",
        "payload": {
            "memory_set": "Adversary" if self.memory_cache["adversary"] else "Self",
            "fidelity_noise_level": self.memory_fidelity_noise_factor,
            "low_awareness_flag": low_awareness_flag,
        },
    }
    self._queue_action("ACTION_ANALYZE_MEMORY", payload)
    reconstructed: List[float] = []
    if self.memory_cache["self"]:
        latest = self.memory_cache["self"][-1].get("lambdaMemory")
        if latest:
            reconstructed = self.memory_api.deserialize_memory(latest).tolist()
    meta = {
        "memorySummaries": {
            "self": len(self.memory_cache["self"]),
            "adversary": len(self.memory_cache["adversary"]),
        },
        "prompt": payload,
        "reconstructedCanonical": reconstructed[:10],
    }
    self.memory_noise_records.append(meta)
    return meta

def mutate_params(self, mode: str) -> None:
    payload = {
        "action": "MUTATE_DDS_HEURISTICS",
        "payload": {
            "target_group": [
                "analysis_weights",
                "rule_thresholds",
                "ligand_design_rates",
            ],
            "scale": (
                self.annealing_mutation_scale_factor_unstable
                if mode == "explore"
                else self.annealing_mutation_scale_factor_stable
            ),
        },
    }
    self._queue_action("ACTION_MUTATE_PARAMS", payload)
```

```
def _register_action_event(self, action: str, agent_name: str) -> None:
    now = time.time()
    self.analysis_action_window.append((now, action, agent_name))
    while self.analysis_action_window and (
        now - self.analysis_action_window[0][0]
    ) > self.zeno_effect_time_window:
        self.analysis_action_window.popleft()
    count = sum(1 for _, act, _ in self.analysis_action_window if act == action)
    self.zeno_dampening_active = count >= self.zeno_effect_trigger_count
    self.action_log.append({"action": action, "agent": agent_name, "timestamp": now})

def _queue_action(self, action_type: str, payload: Dict[str, Any]) -> None:
    entry = {"action": action_type, "timestamp": time.time(), "payload": payload}
    self.action_log.append(entry)
    self.pending_blackboard_posts.append(entry)

def _estimate_awareness_change(self, agent_name: str, report: Dict[str, Any]) -> float:
    if agent_name == "LigandDiscoveryAgent":
        affinity = None
        if isinstance(report, dict):
            affinity = report.get("affinity", {}).get("bindingFreeEnergy")
        if affinity is not None:
            reference = self.quantum_reference.get("statistics", {}).get("meanEnergy", -8.0)
            return 0.06 if affinity < reference else -0.02
        return 0.0
    if agent_name == "QuantumSimulationAgent":
        affinity = None
        if isinstance(report, dict):
            affinity = report.get("affinity", {}).get("bindingFreeEnergy")
        if affinity is not None:
            reference = self.quantum_reference.get("statistics", {}).get("meanEnergy", -8.0)
            return 0.05 if affinity < reference else 0.01
        return 0.0
    if agent_name == "SafetyAgent":
        toxicity = None
        if isinstance(report, dict):
            toxicity = report.get("admet", {}).get("toxicityRiskScore")
        if toxicity is not None:
            return -0.06 if toxicity > 0.25 else 0.03
        return 0.0
    return 0.015

def _apply_superposition(self, agent_name: str, report: Dict[str, Any]) -> Dict[str, Any]:
    meta: Dict[str, Any] = {}
```

```
        if agent_name == "LigandDiscoveryAgent" and isinstance(report, dict):
            tolerance = float(0.1 + 0.2 * self.state_potential)
            affinity_report = report.get("affinity", {})
            reference_mean = self.quantum_reference.get("statistics", {}).get("meanEnergy")
            credible = None
            if isinstance(affinity_report, dict):
                credible = affinity_report.get("bindingFreeEnergyUncertainty",
{}).get("credibleInterval")
            accepted = False
            if reference_mean is not None and credible:
                accepted = bool(credible[0] <= reference_mean <= credible[1])
                affinity_report["superpositionTolerance"] = {
                    "statePotential": self.state_potential,
                    "tolerance": tolerance,
                    "referenceWithinInterval": accepted,
                }
            meta = {
                "statePotential": self.state_potential,
                "tolerance": tolerance,
                "referenceWithinInterval": accepted,
            }
        if agent_name == "JobStatusAgent" and isinstance(report, dict):
            priority = float(1.0 + max(0.0, self.state_potential - 0.5) * 0.5)
            report.setdefault("resourceUtilization", {})["fidelityPriority"] = priority
            meta = {"fidelityPriority": priority, "statePotential": self.state_potential}
        return meta

    def _apply_entanglement_feedback(
        self,
        agent_name: str,
        awareness_delta: float,
    ) -> Dict[str, Any]:
        if abs(awareness_delta) < 0.05:
            return {}
        target = "SafetyAgent" if awareness_delta < 0 else "LigandDiscoveryAgent"
        record = {
            "target": target,
            "learningRateScale": self.entanglement_param_boost_factor,
            "awarenessChange": awareness_delta,
        }
        self.entanglement_records.append(record)
        return record

    def _update_awareness(self, delta: float) -> float:
```

```
        boost = 1.0
        if self.state_potential > 0.5:
            boost += self.potential_awareness_boost_factor
        if self.zeno_dampening_active:
            boost *= self.zeno_effect_dampening_factor
        adjusted = delta * boost
        self.self_awareness = float(np.clip(self.self_awareness + adjusted, 0.0, 1.0))
        self.awareness_history.append(self.self_awareness)
        self.awareness_changes.append(adjusted)
        return adjusted

    def _apply_adaptive_annealing(self) -> Dict[str, Any]:
        if len(self.awareness_history) < 5:
            return {}
        window = list(self.awareness_history)[-20:]
        std = statistics.pstdev(window) if len(window) > 1 else 0.0
        mode = "exploit" if std <= self.annealing_stability_threshold else "explore"
        lr_scale = (
            self.annealing_lr_factor_stable
            if mode == "exploit"
            else self.annealing_lr_factor_unstable
        )
        mutation_scale = (
            self.annealing_mutation_scale_factor_stable
            if mode == "exploit"
            else self.annealing_mutation_scale_factor_unstable
        )
        record = {
            "mode": mode,
            "awarenessStd": std,
            "learningRateScale": lr_scale,
            "mutationScale": mutation_scale,
        }
        if not self.annealing_records or self.annealing_records[-1] != record:
            self.annealing_records.append(record)
            if mode == "explore":
                self.mutate_params(mode)
        return record

    def _check_tunneling_conditions(self) -> Dict[str, Any]:
        if self.stagnation_counter >= 5 or self.pains_alert_counter >= 3:
            if self.self_awareness >= 0.7 and self.state_potential >= 0.3 and self.state_resonance <
0.4:
                return self._perform_quantum_tunnel()
```

```
        return {}

    def _perform_quantum_tunnel(self) -> Dict[str, Any]:
        noise = np.random.normal(0, self.tunneling_state_shift_factor, size=getattr(self.core_ai,
"state_dim", 32))
        if hasattr(self.core_ai, "state"):
            self.core_ai.state = np.asarray(self.core_ai.state, dtype=float) + noise
        new_focus = f"{self.current_target_focus}-lambda-shift-{len(self.tunneling_records) + 1}"
        payload = {
            "reason": "Awareness Stagnation detected; low Resonance Score in state space.",
            "shift_magnitude": self.tunneling_state_shift_factor,
            "new_hypothesis_directive": "Prioritize scaffold hopping into lambda-toroidal sites.",
        }
        self.current_target_focus = new_focus
        self.tunneling_records.append({"newFocus": new_focus, "rewardBonus": 0.5, "payload":
payload})
        self._queue_action("ACTION_QUANTUM_TUNNEL", payload)
        return {"newFocus": new_focus, "rewardBonus": 0.5, "payload": payload}

    def _perform_state_blend(self) -> Dict[str, Any]:
        random_state = np.random.normal(0, 1.0, size=getattr(self.core_ai, "state_dim", 32))
        current = np.asarray(getattr(self.core_ai, "state", np.zeros_like(random_state)),
dtype=float)
        diff_norm = float(np.linalg.norm(current - random_state))
        scale = min(1.0, diff_norm / max(1.0, np.sqrt(random_state.size)))
        blend_factor = self.interference_blend_factor_base * (1.0 + scale)
        blended = (1 - blend_factor) * current + blend_factor * random_state
        if hasattr(self.core_ai, "state"):
            self.core_ai.state = blended
        payload = {
            "differenceNorm": diff_norm,
            "blendFactor": blend_factor,
        }
        self.blend_records.append(payload)
        self._queue_action("ACTION_BLEND_STATE", payload)
        return payload

    def _apply_memory_fidelity_noise(
        self,
        agent_name: str,
        report: Dict[str, Any],
    ) -> Dict[str, Any]:
        noise_scale = self.memory_fidelity_noise_factor * (
            1.0 + max(0.0, 0.5 - self.self_awareness) * 5.0
```

```
        )
        if noise_scale <= 0:
            return {}
        applied = False
        if isinstance(report, dict):
            target_keys: List[Tuple[Dict[str, Any], str]] = []
            if agent_name == "SafetyAgent":
                target_keys.append((report.get("admet", {}), "toxicityRiskScore"))
            elif agent_name == "LigandDiscoveryAgent":
                target_keys.append((report.get("affinity", {}), "bindingFreeEnergy"))
            elif agent_name == "QuantumSimulationAgent":
                target_keys.append((report.get("affinity", {}), "bindingFreeEnergy"))
            for container, key in target_keys:
                if isinstance(container, dict) and key in container:
                    memory_tag = container.setdefault(f"{key}MemoryInflation", {})
                    memory_tag["noiseScale"] = noise_scale
                    memory_tag["awareness"] = self.self_awareness
                    applied = True
        if applied:
            record = {"agent": agent_name, "noiseScale": noise_scale}
            self.memory_noise_records.append(record)
            return record
        return {}

    def _record_binding_score(self, agent_name: str, report: Dict[str, Any]) -> None:
        affinity = None
        if isinstance(report, dict):
            affinity = report.get("affinity", {}).get("bindingFreeEnergy")
        if affinity is None:
            return
        if self.binding_history and abs(affinity - min(self.binding_history)) < 0.05:
            self.stagnation_counter += 1
        else:
            self.stagnation_counter = max(0, self.stagnation_counter - 1)
        self.binding_history.append(float(affinity))

    def _track_pains_alerts(self, report: Dict[str, Any]) -> None:
        admet = report.get("admet") if isinstance(report, dict) else None
        if not isinstance(admet, dict):
            return
        if any(alert == "PAINS" for alert in admet.get("alerts", [])):
            self.pains_alert_counter += 1

    def _register_memory_snapshot(self, agent_name: str, report: Dict[str, Any]) -> None:
```

```
    summary = list(report.keys()) if isinstance(report, dict) else []
    entry = {"agent": agent_name, "summary": summary}
    target = "adversary" if agent_name == "SafetyAgent" else "self"
    agent_ref = self.agent_map.get(agent_name)
    if agent_ref and agent_ref.observation_state.get("lambdaLatentVector") is not None:
        latent_vector = agent_ref.observation_state.get("lambdaLatentVector", [])
        serialized = self.memory_api.serialize_memory(
            latent_vector,
            metadata={"agent": agent_name, "summary": summary, "scaling_basis": "lambda"},
        )
        entry["lambdaMemory"] = serialized
    self.memory_cache[target].append(entry)

async def flush_blackboard(self) -> None:
    while self.pending_blackboard_posts:
        entry = self.pending_blackboard_posts.pop(0)
        payload = {
            "reportId": f"gtai-action-{len(self.action_log)}",
            "entry": entry,
        }
        await self.blackboard.post("gtaiActions", payload)

def compile_summary(self) -> Dict[str, Any]:
    awareness_std = (
        statistics.pstdev(self.awareness_history)
        if len(self.awareness_history) > 1
        else 0.0
    )
    return {
        "state": {
            "statePotential": self.state_potential,
            "selfAwareness": self.self_awareness,
            "stateResonance": self.state_resonance,
            "awarenessStd": awareness_std,
            "awarenessHistoryTail": list(self.awareness_history)[-10:],
        },
        "actions": self.action_log,
        "entanglement": self.entanglement_records,
        "annealing": self.annealing_records,
        "tunneling": self.tunneling_records,
        "interference": self.blend_records,
        "simulations": self.simulation_reflections,
        "metaAnalysis": self.meta_tuning_records,
        "memory": self.memory_noise_records,
```

```
        }

# ---------------------------------------------------------------------------
# Agent definitions
# ---------------------------------------------------------------------------


class AgentBase:
    def __init__(
        self,
        name: str,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
        **kwargs: Any,
    ):
        self.name = name
        self.blackboard = blackboard
        self.context = context
        self.validator = validator
        self.ml_registry = ml_registry
        self.dataset_manager = dataset_manager
        self.feature_extractor = feature_extractor
        self.active_learning = active_learning
        self.ml_api = ml_api
        self.kwargs = kwargs
        self.observation_state: Dict[str, Any] = {}

    async def run(self) -> Dict[str, Any]:
        raise NotImplementedError

    def encode_lambda_latent(self, state: Dict[str, Any]) -> np.ndarray:
        """Convert lambda-shell descriptors into a tensorial latent."""

        descriptors: List[Dict[str, Any]] = []
        if "descriptors" in state:
            descriptors = list(state.get("descriptors", []))
        elif "lambdaShellDiagnostics" in state:
            descriptors = list(state["lambdaShellDiagnostics"].get("descriptors", []))
```

```
        elif state.get("lambdaShellDiagnostics"):
            diag = state.get("lambdaShellDiagnostics")
            descriptors = list(diag.get("descriptors", []))
        if not descriptors:
            descriptors = list(self.context.lambda_shells)
        tensor_rows: List[List[float]] = []
        for descriptor in descriptors or []:
            tensor_rows.append(
                [
                    float(descriptor.get("lambdaRadius", 0.0)),
                    float(descriptor.get("lambdaCurvature", 0.0)),
                    float(descriptor.get("lambdaEntropy", 0.0)),
                    float(descriptor.get("lambdaEnergyDensity", 0.0)),
                    float(descriptor.get("lambdaBhattacharyya", 0.0)),
                    float(descriptor.get("lambdaOccupancy", 0.0)),
                    float(descriptor.get("lambdaLeakage", 0.0)),
                ]
            )
        if not tensor_rows:
            tensor_rows = [[0.0] * 7]
        tensor = np.asarray(tensor_rows, dtype=float)
        self.observation_state["lambdaLatentTensor"] = tensor
        self.observation_state["lambdaLatentVector"] = tensor.flatten()
        return tensor


class StructuralAnalysisAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        data_client: PublicDataClient,
        pdb_id: str,
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "StructuralAnalysisAgent",
            blackboard,
            context,
```

```
        validator,
        ml_registry=ml_registry,
        dataset_manager=dataset_manager,
        feature_extractor=feature_extractor,
        active_learning=active_learning,
        ml_api=ml_api,
    )
    self.data_client = data_client
    self.pdb_id = pdb_id

def embed_lambda_shells(self, molecule_structure: np.ndarray) -> Dict[str, Any]:
    if molecule_structure.size == 0:
        return {"descriptors": [], "tensor": np.zeros((1, 7)), "shellFeatures": {}}
    center = np.mean(molecule_structure, axis=0)
    distances = np.linalg.norm(molecule_structure - center, axis=1)
    shell_indices = (distances / max(LAMBDA_DILATION, 1e-6)).astype(int)
    max_distance = float(np.max(distances) or 1.0)
    shell_features: Dict[str, Dict[str, float]] = {}
    for atom_idx, shell in enumerate(shell_indices.tolist()):
        distance = float(distances[atom_idx])
        curvature = float(
            self.context.curvature_profile[shell % len(self.context.curvature_profile)]
            if self.context.curvature_profile
            else 0.0
        )
        entropy = float(
            self.context.entanglement_entropy
            * (1.0 + 0.05 * shell)
            * np.exp(-distance / max_distance)
        )
        energy_mode = (
            self.context.lambda_modes[shell % len(self.context.lambda_modes)]
            if self.context.lambda_modes
            else 1.0
        )
        energy = float(energy_mode / max(distance, 1e-3))
        key = str(shell)
        bucket = shell_features.setdefault(
            key,
            {"count": 0.0, "shell_curvature": 0.0, "entanglement_density": 0.0, "local_energy": 0.0},
        )
        bucket["count"] += 1.0
        bucket["shell_curvature"] += curvature
        bucket["entanglement_density"] += entropy
```

```
            bucket["local_energy"] += energy
        descriptors: List[Dict[str, Any]] = []
        for shell_key, stats in sorted(shell_features.items(), key=lambda item: int(item[0])):
            shell = int(shell_key)
            count = max(stats.pop("count", 1.0), 1.0)
            stats["shell_curvature"] /= count
            stats["entanglement_density"] /= count
            stats["local_energy"] /= count
            descriptors.append(
                {
                    "shellIndex": shell,
                    "lambdaRadius": max_distance / math.pow(LAMBDA_DILATION, shell),
                    "lambdaCurvature": stats["shell_curvature"],
                    "lambdaEntropy": stats["entanglement_density"],
                    "lambdaEnergyDensity": stats["local_energy"],
                    "lambdaBhattacharyya": float(np.clip(stats["entanglement_density"] /
(self.context.entanglement_entropy + 1e-6), 0.0, 2.0)),
                    "lambdaOccupancy": float(np.clip(count / len(distances), 0.0, 1.0)),
                    "lambdaLeakage": float(np.clip(abs(count / len(distances) - 1.0 / (shell + 1)), 0.0,
1.0)),
                }
            )
        tensor = self.encode_lambda_latent({"descriptors": descriptors})
        embedding = {"descriptors": descriptors, "tensor": tensor, "shellFeatures": shell_features}
        self.observation_state["lambdaShellEmbedding"] = tensor
        self.observation_state["lambdaShellFeatures"] = descriptors
        return embedding

    def _parse_atoms(self, pdb_text: str) -> np.ndarray:
        coords: List[Tuple[float, float, float]] = []
        for line in pdb_text.splitlines():
            if line.startswith("ATOM"):
                try:
                    x = float(line[30:38])
                    y = float(line[38:46])
                    z = float(line[46:54])
                    coords.append((x, y, z))
                except ValueError:
                    continue
        if not coords:
            coords.append((0.0, 0.0, 0.0))
        return np.array(coords)

    def _detect_pockets(
```

```
        self,
        coords: np.ndarray,
        shell_embedding: Optional[Dict[str, Any]] = None,
    ) -> List[Dict[str, Any]]:
        center = np.mean(coords, axis=0)
        distances = np.linalg.norm(coords - center, axis=1)
        threshold = np.percentile(distances, 60)
        pocket_atoms = coords[distances < threshold]
        volume = float(np.ptp(pocket_atoms[:, 0]) * np.ptp(pocket_atoms[:, 1]) *
np.ptp(pocket_atoms[:, 2]) or 1.0)
        hydrophobicity = float(np.clip(0.5 + 0.1 * random.random(), 0.0, 1.0))
        electrostatic = float(-1.0 * (1.0 + 0.1 * random.random()))
        curvature_bias = statistics.mean(self.context.curvature_profile[:10]) if
self.context.curvature_profile else -1.0
        if shell_embedding and shell_embedding.get("shellFeatures"):
            inner_shell = shell_embedding["shellFeatures"].get("0")
            if inner_shell:
                hydrophobicity = float(np.clip(hydrophobicity + 0.05 *
inner_shell.get("entanglement_density", 0.0), 0.0, 1.0))
                curvature_bias = float(inner_shell.get("shell_curvature", curvature_bias))
        druggability = float(np.clip(0.7 + 0.05 * hydrophobicity + 0.01 * curvature_bias, 0.0, 1.0))
        return [
            {
                "pocketId": "pocket-01",
                "druggabilityScore": druggability,
                "rank": 1,
                "properties": {
                    "size": volume,
                    "shape": "ellipsoidal",
                    "volume": volume,
                    "hydrophobicity": hydrophobicity,
                    "electrostaticPotential": electrostatic,
                    "residueComposition": ["LEU:45", "VAL:48", "TYR:88"],
                },
            }
        ]

    def _classify_waters(self, coords: np.ndarray) -> List[Dict[str, Any]]:
        rng = np.random.default_rng(42)
        entries = []
        for idx in range(3):
            classification = rng.choice(["displaceable", "bridging", "stabilizing"], p=[0.4, 0.4, 0.2])
            entry: Dict[str, Any] = {"waterId": f"HOH-{300+idx}", "classification": classification}
            if classification == "displaceable":
```

```
            entry["displacementEnergy"] = float(-2.0 + rng.random())
        else:
            partners = rng.choice(["ASP:25", "LIG:C4", "GLU:120", "HOH:410"], size=2,
replace=False)
            entry["bridgingPartners"] = partners.tolist()
        entries.append(entry)
    return entries

  async def run(self) -> Dict[str, Any]:
    pdb_text = self.data_client.fetch_pdb(self.pdb_id)
    coords = self._parse_atoms(pdb_text)
    lambda_embedding = self.embed_lambda_shells(coords)
    pockets = self._detect_pockets(coords, lambda_embedding)
    waters = self._classify_waters(coords)
    lambda_analysis = LambdaScalingToolkit.analyze_coordinates(coords, self.context)
    lambda_analysis.setdefault("summary", {})["scalingBasis"] = "lambda"
    quantum_patterns = {
        "lambdaCurvatureMean": float(np.mean(self.context.curvature_profile[:20] or [0.0])),
        "emergingSiteClasses": ["lambda-toroidal", "quantum-anchored hydrophobic cavity"],
        "patternRationale": "Patterns mined using quantum-enhanced pocket analysis per
roadmap directive.",
        "lambdaShellAnalysis": {
            "descriptors": lambda_analysis["descriptors"],
            "summary": {**lambda_analysis["summary"], **self.context.lambda_basis},
        },
        "lambdaShellEmbedding": {
            "tensor": lambda_embedding["tensor"].tolist(),
            "descriptors": lambda_embedding["descriptors"],
        },
    }
    ml_section: Dict[str, Any] = {}
    if self.feature_extractor and self.dataset_manager and self.ml_registry:
        task_name = "structural.druggability"
        features_list: List[np.ndarray] = []
        metadata_bundle: List[Dict[str, Any]] = []
        for pocket in pockets:
            feat = self.feature_extractor.featurize_pocket(pocket)
            pocket.setdefault("lambdaShellDiagnostics", lambda_analysis)
            pocket["lambdaShellEmbedding"] = {
                "tensor": lambda_embedding["tensor"].tolist(),
                "descriptors": lambda_embedding["descriptors"],
                "shellFeatures": lambda_embedding["shellFeatures"],
            }
```

```
            lambda_feat = self.feature_extractor.featurize_lambda_shells(lambda_analysis["descriptors"])
            combined_feat = self.feature_extractor.combine_features(feat, lambda_feat)
            features_list.append(combined_feat)
            metadata = {"pocketId": pocket.get("pocketId"), "source": "structural_analysis"}
            metadata_bundle.append(metadata)
            self.dataset_manager.register_record(
                task_name,
                combined_feat,
                pocket.get("druggabilityScore", 0.0),
                {**metadata, "reportId": f"pocket-rep-{self.pdb_id}"},
            )
        split = self.dataset_manager.build_split(task_name)
        model = self.ml_registry.get_model(task_name)
        if model is None and split.train_X.size:
            model = SimpleRegressor(f"{task_name}-reg", architecture="GradientBoostedProxy")
            metrics = model.train(split)
            self.ml_registry.register_model(task_name, model, metrics, split.metadata)
            training_record = lambda_shell_training_hook(
                self.name,
                self.context,
                lambda_analysis,
            )
            self.observation_state.setdefault("trainingLogs", []).append(training_record)
        if model and features_list:
            normalization = split.normalization if split else {"mean": np.zeros_like(features_list[0]), "std": np.ones_like(features_list[0])}
            stacked = np.stack(features_list)
            norm = (
                (stacked - normalization["mean"]) / normalization["std"]
                if isinstance(normalization, dict)
                else stacked
            )
            preds, uncert = model.predict_with_uncertainty(norm)
            ml_section = {
                "task": task_name,
                "model": model.describe(),
                "dataset": split.metadata if split else {"records": 0},
                "predictions": [
                    {
                        "pocketId": meta["pocketId"],
                        "mlDruggability": float(pred),
                        "uncertainty": float(unc),
                    }
```

```
                    for meta, pred, unc in zip(metadata_bundle, preds, uncert)
                ],
            }
            if self.active_learning:
                self.active_learning.evaluate_samples(task_name, features_list, preds, uncert,
metadata_bundle)
            if self.ml_api:
                self.ml_api.register_endpoint("structural/druggability", task_name, model.version)
                self.ml_api.log_call(
                    "structural/druggability",
                    {"pockets": [meta["pocketId"] for meta in metadata_bundle]},
                    {"predictions": ml_section["predictions"]},
                )
        report = {
            "reportId": f"pocket-rep-{self.pdb_id}",
            "sourcePdbId": self.pdb_id,
            "pockets": pockets,
            "waterAnalysis": {"waterMolecules": waters},
            "quantumPocketInsights": quantum_patterns,
            "mlAugmentation": ml_section,
        }
        validated = self.validator.validate(self.name, report)
        await self.blackboard.post("binding", validated)
        return validated


class LigandDiscoveryAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        data_client: PublicDataClient,
        llm: LightweightLLM,
        target_query: str,
        quantum_reference: Dict[str, Any],
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "LigandDiscoveryAgent",
```

```
            blackboard,
            context,
            validator,
            ml_registry=ml_registry,
            dataset_manager=dataset_manager,
            feature_extractor=feature_extractor,
            active_learning=active_learning,
            ml_api=ml_api,
        )
        self.data_client = data_client
        self.llm = llm
        self.target_query = target_query
        self.quantum_reference = quantum_reference

    def compute_shell_entropy_curvature_map(self, ligand: Dict[str, Any]) -> Dict[str, Any]:
        diagnostics = ligand.get("lambdaShellDiagnostics") or {}
        descriptors = diagnostics.get("descriptors", []) or self.context.lambda_shells
        if not descriptors:
            descriptors = []
        curvature_vector: List[float] = []
        entropy_vector: List[float] = []
        shell_map: List[Dict[str, Any]] = []
        for descriptor in descriptors:
            curvature = float(descriptor.get("lambdaCurvature", 0.0))
            entropy = float(descriptor.get("lambdaEntropy", self.context.entanglement_entropy))
            shell_index = int(descriptor.get("shellIndex", len(shell_map)))
            curvature_vector.append(curvature)
            entropy_vector.append(entropy)
            shell_map.append(
                {
                    "shellIndex": shell_index,
                    "curvature": curvature,
                    "entropy": entropy,
                    "curvatureEntropyProduct": curvature * entropy,
                }
            )
        tensor = self.encode_lambda_latent({"descriptors": descriptors})
        map_payload = {
            "shellMap": shell_map,
            "curvatureVector": curvature_vector,
            "entropyVector": entropy_vector,
            "tensor": tensor.tolist(),
        }
        ligand["lambdaShellMap"] = map_payload
```

```
        self.observation_state.setdefault("ligandShellMaps", []).append(map_payload)
        return map_payload

    def _inverse_design(self, pocket_id: str) -> Dict[str, Any]:
        candidates = self.data_client.fetch_pubchem_candidates(self.target_query)
        top = candidates[0]
        return {
            "setId": f"lig-set-{pocket_id}-inverse",
            "sourcePocketId": pocket_id,
            "strategy": "inverse_design",
            "candidates": [
                {
                    "ligandId": top["ligandId"],
                    "smiles": top["smiles"],
                    "drugLikeness": float(np.clip(0.7 + 0.1 * random.random(), 0.0, 1.0)),
                    "syntheticAccessibility": float(np.clip(2.0 + random.random(), 1.0, 10.0)),
                }
            ],
        }

    def _scaffold_hopping(self, pocket_id: str) -> Dict[str, Any]:
        prompt = textwrap.dedent(
            f"""
            Target pocket {pocket_id} exhibits hydrophobic residues and entanglement entropy
{self.context.entanglement_entropy:.3f}.
            Suggest a novel aromatic scaffold with heteroatom donors suitable for COX-2 inhibition.
            """
        ).strip()
        suggestion = self.llm.complete(prompt)
        novelty = float(np.clip(0.85 + 0.05 * random.random(), 0.0, 1.0))
        return {
            "setId": f"lig-set-{pocket_id}-scaffold",
            "sourcePocketId": pocket_id,
            "strategy": "scaffold_hopping",
            "candidates": [
                {
                    "ligandId": "lig-novel-001",
                    "smiles": "c1ccc(NC(=O)NC2=NC=CC=C2)cc1",
                    "noveltyScore": novelty,
                    "predictedInteractions": ["H-bond:TYR:88", "π-π:TRP:120"],
                    "designRationale": suggestion,
                }
            ],
        }
```

```
    async def run(self) -> Dict[str, Any]:
        binding_report = await self.blackboard.read("binding")
        pocket_id = binding_report["pockets"][0]["pocketId"] if binding_report else "pocket-01"
        inverse_set = self._inverse_design(pocket_id)
        scaffold_set = self._scaffold_hopping(pocket_id)
        synthetic_library = []
        ref_samples = self.quantum_reference.get("samples", [])
        for idx, sample in enumerate(ref_samples[:3]):
            synthetic_library.append(
                {
                    "libraryLigandId": f"quant-lib-{idx}",
                    "bindingEnergyRef": sample["bindingEnergy"],
                    "entanglementEntropyRef": sample["entanglementEntropy"],
                }
            )
        motif_alerts = []
        for candidate in scaffold_set["candidates"]:
            if "NN" in candidate["smiles"] or candidate["smiles"].count("N") > 3:
                motif_alerts.append(
                    {
                        "ligandId": candidate["ligandId"],
                        "issue": "Potential unphysical multi-azide motif detected",
                        "recommendedCorrection": "Reduce adjacent nitrogen donors to maintain
stability",
                    }
                )
        combined = {
            "inverse": inverse_set,
            "scaffold": scaffold_set,
            "syntheticLibrary": synthetic_library,
            "motifAlerts": motif_alerts,
            "lambdaShellBasis": self.context.lambda_basis,
        }
        ligand_lambda_diag = None
        for bundle in (inverse_set["candidates"], scaffold_set["candidates"]):
            for candidate in bundle:
                lambda_diag = LambdaScalingToolkit.analyze_ligand(
                    candidate.get("smiles", ""), self.context, self.quantum_reference
                )
                candidate["lambdaShellDiagnostics"] = lambda_diag
                candidate["lambdaShellFingerprint"] = lambda_diag.get("summary", {}).get(
                    "lambdaShellFingerprint", lambda_diag.get("fingerprint", [])
                )
```

```
            base_rationale = candidate.get("designRationale", "").strip()
            candidate["designRationale"] = (
                f"{base_rationale} | λ-shell alignment maintained"
                if base_rationale
                else "λ-shell alignment maintained"
            )
            self.compute_shell_entropy_curvature_map(candidate)
            if ligand_lambda_diag is None:
                ligand_lambda_diag = lambda_diag
        ml_section: Dict[str, Any] = {}
        all_candidates = inverse_set["candidates"] + scaffold_set["candidates"]
        if self.feature_extractor and self.dataset_manager and self.ml_registry and all_candidates:
            task_name = "ligand.bindingAffinity"
            features = []
            metadata_bundle = []
            for candidate in all_candidates:
                feat = self.feature_extractor.featurize_smiles(candidate.get("smiles", ""))
                lambda_feat = self.feature_extractor.featurize_lambda_shells(
                    candidate.get("lambdaShellDiagnostics", {}).get("descriptors", [])
                )
                shell_tensor = np.asarray(candidate.get("lambdaShellMap", {}).get("tensor", []),
dtype=float).flatten()
                combined_feat = self.feature_extractor.combine_features(feat, lambda_feat)
                if shell_tensor.size:
                    combined_feat = self.feature_extractor.combine_features(combined_feat,
shell_tensor)
                features.append(combined_feat)
                metadata = {
                    "ligandId": candidate.get("ligandId"),
                    "source": "ligand_agent",
                }
                metadata_bundle.append(metadata)
                label = candidate.get("drugLikeness") or candidate.get("noveltyScore") or 0.5
                self.dataset_manager.register_record(
                    task_name, combined_feat, label, {**metadata, "labelType": "heuristic"}
                )
            split = self.dataset_manager.build_split(task_name)
            model = self.ml_registry.get_model(task_name)
            if model is None and split.train_X.size:
                model = GraphSurrogateModel(f"{task_name}-gnn")
                metrics = model.train(split)
                self.ml_registry.register_model(task_name, model, metrics, split.metadata)
                training_record = lambda_shell_training_hook(
                    self.name,
```

```
                self.context,
                ligand_lambda_diag or {"descriptors": self.context.lambda_shells},
            )
            self.observation_state.setdefault("trainingLogs", []).append(training_record)
        risk_task = "ligand.offTargetRisk"
        if features:
            for candidate, feat in zip(all_candidates, features):
                flag = 1.0 if any(alert.get("issue") for alert in motif_alerts if alert["ligandId"] ==
candidate.get("ligandId")) else 0.0
                self.dataset_manager.register_record(
                    risk_task,
                    feat,
                    flag,
                    {"ligandId": candidate.get("ligandId"), "source": "ligand_agent"},
                )
        risk_split = self.dataset_manager.build_split(risk_task)
        risk_model = self.ml_registry.get_model(risk_task)
        if risk_model is None and risk_split.train_X.size:
            risk_model = SimpleClassifier(f"{risk_task}-clf", architecture="SVMProxy")
            risk_metrics = risk_model.train(risk_split)
            self.ml_registry.register_model(risk_task, risk_model, risk_metrics,
risk_split.metadata)
            training_record = lambda_shell_training_hook(
                f"{self.name}-risk",
                self.context,
                ligand_lambda_diag or {"descriptors": self.context.lambda_shells},
            )
            self.observation_state.setdefault("trainingLogs", []).append(training_record)
        if model and features:
            normalization = split.normalization if split else {"mean": np.zeros_like(features[0]),
"std": np.ones_like(features[0])}
            stacked = np.stack(features)
            norm_features = (
                (stacked - normalization["mean"]) / normalization["std"]
                if isinstance(normalization, dict)
                else stacked
            )
            preds, uncert = model.predict_with_uncertainty(norm_features)
            ranking = sorted(
                [
                    {
                        "ligandId": meta["ligandId"],
                        "predictedAffinityScore": float(pred),
                        "uncertainty": float(unc),
```

```
                    }
                    for meta, pred, unc in zip(metadata_bundle, preds, uncert)
                ],
                key=lambda item: item["predictedAffinityScore"],
                reverse=True,
            )
            risk_predictions: List[Dict[str, Any]] = []
            if risk_model:
                risk_norm = (
                    (stacked - risk_split.normalization["mean"]) / risk_split.normalization["std"]
                    if risk_split and isinstance(risk_split.normalization, dict) and risk_split.train_X.size
                    else norm_features
                )
                risk_scores, risk_unc = risk_model.predict_with_uncertainty(risk_norm)
                risk_predictions = [
                    {
                        "ligandId": meta["ligandId"],
                        "riskScore": float(score),
                        "uncertainty": float(u),
                    }
                    for meta, score, u in zip(metadata_bundle, risk_scores, risk_unc)
                ]
                if self.active_learning:
                    self.active_learning.evaluate_samples(risk_task, features, risk_scores, risk_unc,
metadata_bundle)
            ml_section = {
                "affinityModel": model.describe(),
                "riskModel": risk_model.describe() if risk_model else None,
                "rankedCandidates": ranking,
                "riskPredictions": risk_predictions,
                "datasets": {
                    "affinity": split.metadata if split else {"records": 0},
                    "risk": risk_split.metadata if risk_split else {"records": 0},
                },
            }
            if self.active_learning:
                self.active_learning.evaluate_samples(task_name, features, preds, uncert,
metadata_bundle)
            if self.ml_api:
                self.ml_api.register_endpoint("ligand/affinity", task_name, model.version)
                self.ml_api.log_call(
                    "ligand/affinity",
                    {"ligands": [meta["ligandId"] for meta in metadata_bundle]},
                    {"rankedCandidates": ranking[:3]},
```

```
                )
                if risk_model:
                    self.ml_api.register_endpoint("ligand/risk", risk_task, risk_model.version)
                    self.ml_api.log_call(
                        "ligand/risk",
                        {"ligands": [meta["ligandId"] for meta in metadata_bundle]},
                        {"riskPredictions": risk_predictions},
                    )
        combined["mlAugmentation"] = ml_section
        validated = self.validator.validate(self.name, combined)
        await self.blackboard.post("ligands", validated)
        return validated


class QuantumSimulationAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        uniprot_meta: Dict[str, Any],
        quantum_reference: Dict[str, Any],
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "QuantumSimulationAgent",
            blackboard,
            context,
            validator,
            ml_registry=ml_registry,
            dataset_manager=dataset_manager,
            feature_extractor=feature_extractor,
            active_learning=active_learning,
            ml_api=ml_api,
        )
        self.uniprot_meta = uniprot_meta
        self.quantum_reference = quantum_reference

    def simulate_quantum_state(
        self,
```

```
        binding_site: Optional[Dict[str, Any]],
        ligand_diag: Optional[Dict[str, Any]],
    ) -> Dict[str, Any]:
        epsilon = 0.01
        scales = [1.0, max(LAMBDA_DILATION - epsilon, 0.1), LAMBDA_DILATION + epsilon]
        base_descriptors = []
        if binding_site and binding_site.get("pockets"):
            base_descriptors = binding_site["pockets"][0].get("lambdaShellDiagnostics",
{}).get("descriptors", [])
        if not base_descriptors and ligand_diag:
            base_descriptors = ligand_diag.get("descriptors", [])
        if not base_descriptors:
            base_descriptors = self.context.lambda_shells
        shell_count = max(len(base_descriptors), 1)
        shell_probabilities: List[List[float]] = []
        for scale in scales:
            probs: List[float] = []
            total = 0.0
            for descriptor in base_descriptors:
                occupancy = float(descriptor.get("lambdaOccupancy", 1.0 / shell_count))
                shell_index = float(descriptor.get("shellIndex", len(probs)))
                prob = float(np.clip(occupancy * (scale ** (-shell_index)), 1e-6, 1.0))
                probs.append(prob)
                total += prob
            if total > 0:
                probs = [p / total for p in probs]
            else:
                probs = [1.0 / shell_count] * shell_count
            shell_probabilities.append(probs)
        baseline = shell_probabilities[0]
        overlaps: Dict[str, float] = {}
        bhattacharyya_scores: List[float] = []
        for idx, perturbed in enumerate(shell_probabilities[1:], start=1):
            overlap = float(
                np.mean(
                    [
                        LambdaScalingToolkit._bhattacharyya(base, pert)
                        for base, pert in zip(baseline, perturbed)
                    ]
                )
            )
            overlaps[f"scale_{idx}"] = overlap
            bhattacharyya_scores.append(overlap)
        stability = float(np.mean(bhattacharyya_scores)) if bhattacharyya_scores else 1.0
```

```
        diagnostics = {
            "scales": scales,
            "shellProbabilities": shell_probabilities,
            "bhattacharyyaOverlap": overlaps,
            "lambdaShiftStabilityScore": stability,
        }
        self.observation_state["lambdaShiftDiagnostics"] = diagnostics
        return diagnostics

    def _simulate_pose(self, ligand_id: str, lambda_diag: Optional[Dict[str, Any]] = None) ->
Dict[str, Any]:
        base_affinity = -8.5 - 0.5 * self.context.enhancement_factor
        affinity = base_affinity + random.uniform(-0.8, 0.3)
        shape = float(np.clip(0.7 + 0.1 * random.random(), 0.0, 1.0))
        pose = {
            "poseId": f"pose-{ligand_id}",
            "bindingAffinityScore": affinity,
            "shapeComplementarity": shape,
            "keyInteractions": ["H-bond:SER:530", "salt-bridge:ARG:513"],
            "ensembleStatistics": {
                "canonicalBeta": 1.0 / max(0.1, 298.0 * 0.001987),
                "partitionFunction": float(
                    np.exp(-affinity / max(0.1, abs(self.quantum_reference["statistics"]["meanEnergy"]))
)
                ),
            },
        }
        if lambda_diag:
            pose["lambdaShellDiagnostics"] = lambda_diag
            pose["lambdaShellFingerprint"] = lambda_diag.get("summary", {}).get(
                "lambdaShellFingerprint", lambda_diag.get("fingerprint", [])
            )
        return pose

    def _binding_affinity(
        self, ligand_id: str, pose: Dict[str, Any], lambda_diag: Optional[Dict[str, Any]] = None
    ) -> Dict[str, Any]:
        electro = -5.0 - 0.2 * self.context.enhancement_factor
        dispersion = -4.0 + 0.1 * random.random()
        hydrogen = -3.0 - 0.1 * random.random()
        free_energy = pose["bindingAffinityScore"] - 0.5 * self.context.entanglement_entropy
        reference_energy = self.quantum_reference.get("statistics", {}).get("meanEnergy",
free_energy)
```

```
        energy_bounds = self.quantum_reference.get("statistics", {}).get("energyRange", {"min":
-60.0, "max": -0.5})
        report = {
            "reportId": f"qm-rep-{ligand_id}",
            "sourceLigandId": ligand_id,
            "bindingFreeEnergy": float(free_energy),
            "confidence": float(np.clip(0.85 + 0.05 * random.random(), 0.0, 1.0)),
            "energyDecomposition": {
                "electrostatics": float(electro),
                "dispersion": float(dispersion),
                "hydrogenBonding": float(hydrogen),
            },
            "target": self.uniprot_meta,
            "referenceComparison": {
                "deltaFromQuantumMean": float(free_energy - reference_energy),
                "withinReferenceBounds": bool(energy_bounds["min"] <= free_energy <=
energy_bounds["max"]),
            },
        }
        if lambda_diag:
            report["lambdaShellDiagnostics"] = lambda_diag
            report["lambdaShellFingerprint"] = lambda_diag.get("summary", {}).get(
                "lambdaShellFingerprint", lambda_diag.get("fingerprint", [])
            )
        return report

    async def run(self) -> Dict[str, Any]:
        binding_report = await self.blackboard.read("binding")
        ligand_report = await self.blackboard.read("ligands")
        ligand_id = "lig-novel-001"
        smiles = ""
        if ligand_report and ligand_report.get("inverse", {}).get("candidates"):
            ligand_id = ligand_report["inverse"]["candidates"][0]["ligandId"]
            smiles = ligand_report["inverse"]["candidates"][0].get("smiles", "")
        ligand_lambda = None
        if ligand_report:
            for bucket in ("inverse", "scaffold"):
                candidates = ligand_report.get(bucket, {}).get("candidates", [])
                if candidates and candidates[0].get("lambdaShellDiagnostics"):
                    ligand_lambda = candidates[0]["lambdaShellDiagnostics"]
                    break
        lambda_shift = self.simulate_quantum_state(binding_report, ligand_lambda)
        pose = self._simulate_pose(ligand_id, ligand_lambda)
        pose["lambdaShiftDiagnostics"] = lambda_shift
```

```
        ml_section: Dict[str, Any] = {}
        if self.feature_extractor and self.dataset_manager and self.ml_registry:
            sample = {
                "ligandId": ligand_id,
                "bindingEnergy": pose["bindingAffinityScore"],
                "entanglementEntropy": self.context.entanglement_entropy,
                "orbitalOccupations": self.quantum_reference.get("samples",
[{}])[0].get("orbitalOccupations", [0.25, 0.25, 0.25, 0.25]),
                "lambdaShellFingerprint": (ligand_lambda or {}).get("summary", {}).get(
                    "lambdaShellFingerprint", (ligand_lambda or {}).get("fingerprint", [])
                ),
            }
            quantum_feat = self.feature_extractor.featurize_quantum_sample(sample)
            smiles_feat = self.feature_extractor.featurize_smiles(smiles or ligand_id)
            lambda_feat = self.feature_extractor.featurize_lambda_shells(
                (ligand_lambda or {}).get("descriptors", self.context.lambda_shells)
            )
            feature_vec = self.feature_extractor.combine_features(quantum_feat, smiles_feat,
lambda_feat)
            task_name = "quantum.bindingEnergy"
            self.dataset_manager.register_record(
                task_name,
                feature_vec,
                pose["bindingAffinityScore"],
                {"ligandId": ligand_id, "source": "quantum_agent"},
            )
            split = self.dataset_manager.build_split(task_name)
            model = self.ml_registry.get_model(task_name)
            if model is None and split.train_X.size:
                model = GraphSurrogateModel(f"{task_name}-surrogate")
                metrics = model.train(split)
                self.ml_registry.register_model(task_name, model, metrics, split.metadata)
                training_record = lambda_shell_training_hook(
                    self.name,
                    self.context,
                    ligand_lambda or {"descriptors": self.context.lambda_shells},
                )
                self.observation_state.setdefault("trainingLogs", []).append(training_record)
            high_task = "quantum.highAffinity"
            label = 1.0 if pose["bindingAffinityScore"] < self.quantum_reference.get("statistics",
{}).get("meanEnergy", -8.0) else 0.0
            self.dataset_manager.register_record(
                high_task,
                feature_vec,
```

```
            label,
            {"ligandId": ligand_id, "source": "quantum_agent"},
        )
        high_split = self.dataset_manager.build_split(high_task)
        classifier = self.ml_registry.get_model(high_task)
        if classifier is None and high_split.train_X.size:
            classifier = SimpleClassifier(f"{high_task}-clf", architecture="LogisticProxy")
            metrics = classifier.train(high_split)
            self.ml_registry.register_model(high_task, classifier, metrics, high_split.metadata)
            training_record = lambda_shell_training_hook(
                f"{self.name}-highAffinity",
                self.context,
                ligand_lambda or {"descriptors": self.context.lambda_shells},
            )
            self.observation_state.setdefault("trainingLogs", []).append(training_record)
        if model:
            normalization = split.normalization if split else {"mean": np.zeros_like(feature_vec),
"std": np.ones_like(feature_vec)}
            norm_feat = (
                (feature_vec - normalization["mean"]) / normalization["std"]
                if isinstance(normalization, dict)
                else feature_vec
            )
            preds, uncert = model.predict_with_uncertainty(norm_feat.reshape(1, -1))
            pose["bindingAffinityScore"] = float(np.mean([pose["bindingAffinityScore"], preds[0]]))
            affinity_prediction = float(preds[0])
            uncertainty = float(uncert[0])
            high_pred: Optional[Dict[str, Any]] = None
            if classifier:
                high_norm = (
                    (feature_vec - high_split.normalization["mean"]) / high_split.normalization["std"]
                    if high_split and isinstance(high_split.normalization, dict)
                    else norm_feat
                )
                risk_score, risk_unc = classifier.predict_with_uncertainty(high_norm.reshape(1, -1))
                high_pred = {
                    "highAffinityProbability": float(risk_score[0]),
                    "uncertainty": float(risk_unc[0]),
                }
                if self.active_learning:
                    self.active_learning.evaluate_samples(
                        high_task,
                        [feature_vec],
                        risk_score,
```

```
                    risk_unc,
                    [{"ligandId": ligand_id}],
                )
            ml_section = {
                "bindingEnergyModel": model.describe(),
                "predictedBindingEnergy": affinity_prediction,
                "uncertainty": uncertainty,
                "highAffinity": high_pred,
                "dataset": split.metadata if split else {"records": 0},
            }
            if self.active_learning:
                self.active_learning.evaluate_samples(
                    task_name,
                    [feature_vec],
                    preds,
                    uncert,
                    [{"ligandId": ligand_id}],
                )
            if self.ml_api:
                self.ml_api.register_endpoint("quantum/binding", task_name, model.version)
                self.ml_api.log_call(
                    "quantum/binding",
                    {"ligandId": ligand_id},
                    {"predictedBindingEnergy": affinity_prediction, "uncertainty": uncertainty},
                )
                if classifier:
                    self.ml_api.register_endpoint("quantum/highAffinity", high_task,
classifier.version)
                    self.ml_api.log_call(
                        "quantum/highAffinity",
                        {"ligandId": ligand_id},
                        {"highAffinity": high_pred},
                    )
        docking_report = {
            "reportId": f"dock-rep-{ligand_id}",
            "sourceLigandId": ligand_id,
            "poses": [pose],
        }
        affinity_report = self._binding_affinity(ligand_id, pose, ligand_lambda)
        affinity_report["lambdaShiftStabilityScore"] = lambda_shift.get("lambdaShiftStabilityScore",
1.0)
        if ml_section:
            docking_report["mlAugmentation"] = ml_section
            affinity_report["mlAugmentation"] = ml_section
```

```
        quantum_dataset = {
            "generatedPoseLibrary": [pose],
            "quantumReference": self.quantum_reference,
            "lambdaShellDiagnostics": ligand_lambda,
        }
        validated_docking = self.validator.validate(self.name, docking_report)
        validated_affinity = self.validator.validate(self.name, affinity_report)
        validated_dataset = self.validator.validate(
            self.name,
            {
                "reportId": f"quant-dataset-{ligand_id}",
                "sourceLigandId": ligand_id,
                "dataset": quantum_dataset,
            },
        )
        await self.blackboard.post("docking", validated_docking)
        await self.blackboard.post("quantum", validated_affinity)
        await self.blackboard.post("quantumDataset", validated_dataset)
        return {"docking": validated_docking, "affinity": validated_affinity, "quantumDataset":
validated_dataset}


class SynthesisPlannerAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        llm: LightweightLLM,
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "SynthesisPlannerAgent",
            blackboard,
            context,
            validator,
            ml_registry=ml_registry,
            dataset_manager=dataset_manager,
            feature_extractor=feature_extractor,
            active_learning=active_learning,
```

```
            ml_api=ml_api,
        )
        self.llm = llm

    async def run(self) -> Dict[str, Any]:
        quantum_report = await self.blackboard.read("quantum")
        ligand_report = await self.blackboard.read("ligands")
        ligand_id = quantum_report["sourceLigandId"] if quantum_report else "lig-novel-001"
        ligand_lambda = None
        if ligand_report:
            for bucket in ("scaffold", "inverse"):
                candidates = ligand_report.get(bucket, {}).get("candidates", [])
                if candidates and candidates[0].get("lambdaShellDiagnostics"):
                    ligand_lambda = candidates[0]["lambdaShellDiagnostics"]
                    break
        if ligand_lambda is None and quantum_report and
quantum_report.get("lambdaShellDiagnostics"):
            ligand_lambda = quantum_report.get("lambdaShellDiagnostics")
        prompt = textwrap.dedent(
            f"""
            Design a three-step synthetic route for ligand {ligand_id} optimizing yield under
quantum-enhanced conditions.
            Incorporate λ-scale invariant considerations for solvent and catalyst selection.
            """
        )
        plan_text = self.llm.complete(prompt)
        plan = [step.strip() for step in plan_text.split(".") if step.strip()][:3]
        if not plan:
            plan = [
                "Couple substituted aniline with activated carbamate under microwave conditions",
                "Perform selective nitration guided by entanglement-aligned field gradients",
                "Finalize via hydrogenation with lambda-stabilized palladium catalyst",
            ]
        lambda_latent = self.encode_lambda_latent(ligand_lambda or {"descriptors":
self.context.lambda_shells})
        accessibility = None
        if ligand_report and ligand_report.get("inverse"):
            candidates = ligand_report["inverse"].get("candidates", [])
            if candidates:
                accessibility = candidates[0].get("syntheticAccessibility")
        rationale = "Plan aligns with lambda-scale catalysts and avoids known reactive
intermediates."
        flags = []
        if accessibility and accessibility > 6.5:
```

```
            flags.append(
                {
                    "type": "SyntheticAccessibility",
                    "message": "Route requires optimization to reduce complexity (score > 6.5)",
                }
            )
        report = {
            "reportId": f"syn-rep-{ligand_id}",
            "sourceLigandId": ligand_id,
            "steps": plan,
            "feasibilityAssessment": {
                "syntheticAccessibility": accessibility if accessibility is not None else 5.0,
                "flags": flags,
            },
            "rationale": rationale,
            "lambdaLatentTensor": lambda_latent.tolist(),
        }
        if ligand_lambda:
            report["lambdaShellDiagnostics"] = ligand_lambda
        if self.feature_extractor and self.dataset_manager and self.ml_registry:
            sequence = " ".join(plan)
            feature_vec = self.feature_extractor.featurize_sequence(sequence)
            lambda_feat = self.feature_extractor.featurize_lambda_shells(
                (ligand_lambda or {}).get("descriptors", [])
            )
            combined_vec = self.feature_extractor.combine_features(feature_vec, lambda_feat)
            combined_vec = self.feature_extractor.combine_features(combined_vec,
lambda_latent.flatten())
            task_name = "synthesis.successProbability"
            label = 1.0 if not flags else 0.5
            self.dataset_manager.register_record(
                task_name,
                combined_vec,
                label,
                {"ligandId": ligand_id, "source": "synthesis_agent"},
            )
            split = self.dataset_manager.build_split(task_name)
            model = self.ml_registry.get_model(task_name)
            if model is None and split.train_X.size:
                model = SimpleClassifier(f"{task_name}-clf", architecture="GradientBoostingProxy")
                metrics = model.train(split)
                self.ml_registry.register_model(task_name, model, metrics, split.metadata)
                training_record = lambda_shell_training_hook(
                    self.name,
```

```
                self.context,
                ligand_lambda or {"descriptors": self.context.lambda_shells},
            )
            self.observation_state.setdefault("trainingLogs", []).append(training_record)
        if model:
            normalization = split.normalization if split else {"mean": np.zeros_like(combined_vec),
"std": np.ones_like(combined_vec)}
            norm_vec = (
                (combined_vec - normalization["mean"]) / normalization["std"]
                if isinstance(normalization, dict)
                else combined_vec
            )
            probs, uncert = model.predict_with_uncertainty(norm_vec.reshape(1, -1))
            ml_section = {
                "successProbability": float(probs[0]),
                "uncertainty": float(uncert[0]),
                "model": model.describe(),
                "dataset": split.metadata if split else {"records": 0},
            }
            report["mlAugmentation"] = ml_section
            if self.active_learning:
                self.active_learning.evaluate_samples(
                    task_name,
                    [feature_vec],
                    probs,
                    uncert,
                    [{"ligandId": ligand_id}],
                )
            if self.ml_api:
                self.ml_api.register_endpoint("synthesis/success", task_name, model.version)
                self.ml_api.log_call(
                    "synthesis/success",
                    {"ligandId": ligand_id},
                    {"successProbability": ml_section["successProbability"]},
                )
        validated = self.validator.validate(self.name, report)
        await self.blackboard.post("synthesis", validated)
        return validated


class ScreeningAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
```

```
        context: QuantumContext,
        validator: PhysicalValidator,
        quantum_reference: Dict[str, Any],
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "ScreeningAgent",
            blackboard,
            context,
            validator,
            ml_registry=ml_registry,
            dataset_manager=dataset_manager,
            feature_extractor=feature_extractor,
            active_learning=active_learning,
            ml_api=ml_api,
        )
        self.quantum_reference = quantum_reference

    async def run(self) -> Dict[str, Any]:
        ligand_report = await self.blackboard.read("ligands")
        ligand_id = "lig-novel-001"
        ligand_lambda = None
        if ligand_report and ligand_report.get("scaffold", {}).get("candidates"):
            ligand_id = ligand_report["scaffold"]["candidates"][0]["ligandId"]
            ligand_lambda = ligand_report["scaffold"]["candidates"][0].get("lambdaShellDiagnostics")
        if ligand_lambda is None and ligand_report and ligand_report.get("inverse",
{}).get("candidates"):
            ligand_lambda = ligand_report["inverse"]["candidates"][0].get("lambdaShellDiagnostics")
        entropy_values = [sample["entanglementEntropy"] for sample in
self.quantum_reference.get("samples", [])]
        trend = float(np.mean(entropy_values)) if entropy_values else
float(self.context.entanglement_entropy)
        lambda_latent = self.encode_lambda_latent(ligand_lambda or {"descriptors":
self.context.lambda_shells})
        report = {
            "reportId": f"screen-rep-{ligand_id}",
            "sourceLigandId": ligand_id,
            "pharmacophoreFeatures": ["HBD:1", "HBA:2", "Aro:1"],
            "topHits": [
```

```
            {"hitId": "ZINC12345", "matchingScore": float(np.clip(0.85 + 0.05 * random.random(),
0.0, 1.0))},
            {"hitId": "ZINC98765", "matchingScore": float(np.clip(0.82 + 0.05 * random.random(),
0.0, 1.0))},
        ],
        "pharmacophoreTrendAnalysis": {
            "meanEntanglementEntropy": trend,
            "trendRationale": "Derived from quantum-generated pose libraries to extend beyond
primary screening.",
        },
        "lambdaLatentTensor": lambda_latent.tolist(),
    }
    if ligand_lambda:
        report["lambdaShellAlignment"] = {
            "descriptors": ligand_lambda.get("descriptors", []),
            "summary": ligand_lambda.get("summary", {}),
        }
    if self.feature_extractor and self.dataset_manager and self.ml_registry:
        task_name = "screening.matchingScore"
        features = []
        metadata_bundle = []
        for hit in report["topHits"]:
            lambda_feat = self.feature_extractor.featurize_lambda_shells(
                (ligand_lambda or {}).get("descriptors", [])
            )
            base_vec = np.array([hit["matchingScore"], trend, self.context.enhancement_factor],
dtype=float)
            feature_vec = self.feature_extractor.combine_features(base_vec, lambda_feat)
            feature_vec = self.feature_extractor.combine_features(feature_vec,
lambda_latent.flatten())
            features.append(feature_vec)
            metadata = {"hitId": hit["hitId"], "ligandId": ligand_id}
            metadata_bundle.append(metadata)
            self.dataset_manager.register_record(
                task_name,
                feature_vec,
                hit["matchingScore"],
                {**metadata, "source": "screening_agent"},
            )
        split = self.dataset_manager.build_split(task_name)
        model = self.ml_registry.get_model(task_name)
        if model is None and split.train_X.size:
            model = SimpleRegressor(f"{task_name}-reg", architecture="RandomForestProxy")
            metrics = model.train(split)
```

```
            self.ml_registry.register_model(task_name, model, metrics, split.metadata)
            training_record = lambda_shell_training_hook(
                self.name,
                self.context,
                ligand_lambda or {"descriptors": self.context.lambda_shells},
            )
            self.observation_state.setdefault("trainingLogs", []).append(training_record)
        ml_section: Dict[str, Any] = {}
        if model and features:
            normalization = split.normalization if split else {"mean": np.zeros_like(features[0]),
"std": np.ones_like(features[0])}
            stacked = np.stack(features)
            norm_features = (
                (stacked - normalization["mean"]) / normalization["std"]
                if isinstance(normalization, dict)
                else stacked
            )
            preds, uncert = model.predict_with_uncertainty(norm_features)
            ml_section = {
                "model": model.describe(),
                "predictions": [
                    {
                        "hitId": meta["hitId"],
                        "score": float(pred),
                        "uncertainty": float(u),
                    }
                    for meta, pred, u in zip(metadata_bundle, preds, uncert)
                ],
                "dataset": split.metadata if split else {"records": 0},
            }
            report["mlAugmentation"] = ml_section
            if self.active_learning:
                self.active_learning.evaluate_samples(task_name, features, preds, uncert,
metadata_bundle)
            if self.ml_api:
                self.ml_api.register_endpoint("screening/matching", task_name, model.version)
                self.ml_api.log_call(
                    "screening/matching",
                    {"ligandId": ligand_id},
                    {"predictions": ml_section["predictions"]},
                )
    validated = self.validator.validate(self.name, report)
    await self.blackboard.post("screening", validated)
    return validated
```

```
class SafetyAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        quantum_reference: Dict[str, Any],
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "SafetyAgent",
            blackboard,
            context,
            validator,
            ml_registry=ml_registry,
            dataset_manager=dataset_manager,
            feature_extractor=feature_extractor,
            active_learning=active_learning,
            ml_api=ml_api,
        )
        self.quantum_reference = quantum_reference

    def detect_lambda_anomalies(self, ligand_embedding: np.ndarray) -> Dict[str, Any]:
        reference_tensor = self.encode_lambda_latent({"descriptors": self.context.lambda_shells})
        reference_vector = reference_tensor.flatten()
        ligand_vector = ligand_embedding.flatten() if ligand_embedding.size else
np.zeros_like(reference_vector)
        ref_abs = np.abs(reference_vector) + 1e-9
        ligand_abs = np.abs(ligand_vector) + 1e-9
        ref_prob = ref_abs / np.sum(ref_abs)
        ligand_prob = ligand_abs / np.sum(ligand_abs)
        kl_divergence = float(np.sum(ligand_prob * np.log(ligand_prob / ref_prob)))
        shell_jump = float(np.max(np.abs(np.diff(ligand_prob)))) if ligand_prob.size > 1 else 0.0
        anomaly_flag = bool(kl_divergence > 0.5 or shell_jump > 0.3)
        record = {
            "klDivergence": kl_divergence,
            "maxShellJump": shell_jump,
            "flagged": anomaly_flag,
```

```
        }
        self.observation_state["lambdaLatentTensor"] = ligand_embedding
        self.observation_state["lambdaLatentVector"] = ligand_vector
        self.observation_state.setdefault("lambdaSafetyDiagnostics", []).append(record)
        return record

    async def run(self) -> Dict[str, Any]:
        quantum_report = await self.blackboard.read("quantum")
        ligand_report = await self.blackboard.read("ligands")
        ligand_id = quantum_report["sourceLigandId"] if quantum_report else "lig-novel-001"
        ligand_lambda = None
        if ligand_report:
            for bucket in ("scaffold", "inverse"):
                candidates = ligand_report.get(bucket, {}).get("candidates", [])
                if candidates and candidates[0].get("lambdaShellDiagnostics"):
                    ligand_lambda = candidates[0]["lambdaShellDiagnostics"]
                    break
        if ligand_lambda is None and quantum_report and
quantum_report.get("lambdaShellDiagnostics"):
            ligand_lambda = quantum_report.get("lambdaShellDiagnostics")
        lambda_latent = self.encode_lambda_latent(ligand_lambda or {"descriptors":
self.context.lambda_shells})
        anomaly = self.detect_lambda_anomalies(lambda_latent)
        entropy_values = [sample["entanglementEntropy"] for sample in
self.quantum_reference.get("samples", [])]
        variability = float(np.std(entropy_values)) if entropy_values else 0.1
        admet_report = {
            "reportId": f"admet-rep-{ligand_id}",
            "sourceLigandId": ligand_id,
            "toxicityRiskScore": float(np.clip(0.18 + 0.05 * random.random(), 0.0, 1.0)),
            "alerts": ["PAINS"],
            "pharmacokinetics": {"bioavailability": 0.62, "halfLife": 4.0},
            "rationale": "Quantum-informed ensemble PK modeling across population variability.",
            "populationVariability": {
                "entropyStdDev": variability,
                "ensembleRuns": 128,
            },
            "lambdaAnomaly": anomaly,
            "lambdaLatentTensor": lambda_latent.tolist(),
        }
        if ligand_lambda:
            admet_report["lambdaShellDiagnostics"] = ligand_lambda
        off_target_report = {
            "reportId": f"off-target-rep-{ligand_id}",
```

```
            "sourceLigandId": ligand_id,
            "potentialOffTargets": [
                {"protein": "Kinase-XYZ", "predictedAffinity": float(-7.4 + 0.2 * random.random())},
                {"protein": "GPCR-123", "predictedAffinity": float(-6.9 + 0.2 * random.random())},
            ],
            "rationale": "Screened against ensemble of public GPCR/kinase datasets with quantum
corrections.",
        }
        if "PAINS" in admet_report["alerts"]:
            admet_report.setdefault("flags", []).append(
                {
                    "type": "SafetyAlert",
                    "message": "PAINS pattern detected; requires medicinal chemistry mitigation.",
                }
            )
        ml_section: Dict[str, Any] = {}
        if self.dataset_manager and self.ml_registry:
            task_name = "safety.toxicity"
            lambda_feat = self.feature_extractor.featurize_lambda_shells(
                (ligand_lambda or {}).get("descriptors", [])
            ) if self.feature_extractor else np.zeros(7)
            base_vec = np.array(
                [
                    admet_report["toxicityRiskScore"],
                    admet_report["pharmacokinetics"]["bioavailability"],
                    admet_report["pharmacokinetics"]["halfLife"],
                    variability,
                    anomaly["klDivergence"],
                    anomaly["maxShellJump"],
                ],
                dtype=float,
            )
            feature_vec = self.feature_extractor.combine_features(base_vec, lambda_feat) if
self.feature_extractor else base_vec
            if self.feature_extractor:
                feature_vec = self.feature_extractor.combine_features(feature_vec,
lambda_latent.flatten())
            label = 1.0 if "PAINS" in admet_report["alerts"] else 0.0
            self.dataset_manager.register_record(
                task_name,
                feature_vec,
                label,
                {"ligandId": ligand_id, "source": "safety_agent"},
            )
```

```
        split = self.dataset_manager.build_split(task_name)
        model = self.ml_registry.get_model(task_name)
        if model is None and split.train_X.size:
            model = SimpleClassifier(f"{task_name}-clf", architecture="DeepNNProxy")
            metrics = model.train(split)
            self.ml_registry.register_model(task_name, model, metrics, split.metadata)
            training_record = lambda_shell_training_hook(
                self.name,
                self.context,
                ligand_lambda or {"descriptors": self.context.lambda_shells},
            )
            self.observation_state.setdefault("trainingLogs", []).append(training_record)
        if model:
            normalization = split.normalization if split else {"mean": np.zeros_like(feature_vec),
"std": np.ones_like(feature_vec)}
            norm_vec = (
                (feature_vec - normalization["mean"]) / normalization["std"]
                if isinstance(normalization, dict)
                else feature_vec
            )
            probs, uncert = model.predict_with_uncertainty(norm_vec.reshape(1, -1))
            ml_section = {
                "toxicityProbability": float(probs[0]),
                "uncertainty": float(uncert[0]),
                "model": model.describe(),
                "dataset": split.metadata if split else {"records": 0},
            }
            if self.active_learning:
                self.active_learning.evaluate_samples(
                    task_name,
                    [feature_vec],
                    probs,
                    uncert,
                    [{"ligandId": ligand_id}],
                )
            if self.ml_api:
                self.ml_api.register_endpoint("safety/toxicity", task_name, model.version)
                self.ml_api.log_call(
                    "safety/toxicity",
                    {"ligandId": ligand_id},
                    {"toxicityProbability": ml_section.get("toxicityProbability")},
                )
    payload = {"admet": admet_report, "offTarget": off_target_report, "mlAugmentation":
ml_section}
```

```
        validated = self.validator.validate(self.name, payload)
        await self.blackboard.post("admet", validated)
        return validated


class IPAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        data_client: PublicDataClient,
        query: str,
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "IPAgent",
            blackboard,
            context,
            validator,
            ml_registry=ml_registry,
            dataset_manager=dataset_manager,
            feature_extractor=feature_extractor,
            active_learning=active_learning,
            ml_api=ml_api,
        )
        self.data_client = data_client
        self.query = query

    async def run(self) -> Dict[str, Any]:
        quantum_report = await self.blackboard.read("quantum")
        ligand_report = await self.blackboard.read("ligands")
        ligand_id = quantum_report["sourceLigandId"] if quantum_report else "lig-novel-001"
        ligand_lambda = None
        if ligand_report:
            for bucket in ("scaffold", "inverse"):
                candidates = ligand_report.get(bucket, {}).get("candidates", [])
                if candidates and candidates[0].get("lambdaShellDiagnostics"):
                    ligand_lambda = candidates[0]["lambdaShellDiagnostics"]
                    break
```

```
        hits = self.data_client.fetch_patent_hits(self.query)
        lambda_latent = self.encode_lambda_latent(ligand_lambda or {"descriptors":
self.context.lambda_shells})
        report = {
            "reportId": f"ip-rep-{ligand_id}",
            "sourceLigandId": ligand_id,
            "noveltyAssessment": "High",
            "freedomToOperateRisk": "Low",
            "conflictingPatents": hits,
            "generativeGraphCrosslinks": [
                {
                    "graphId": "gen-graph-001",
                    "description": "Quantum-derived scaffold library cross-linked with WIPO dataset",
                }
            ],
            "lambdaLatentTensor": lambda_latent.tolist(),
        }
        if ligand_lambda:
            report["lambdaShellDiagnostics"] = ligand_lambda
        if self.dataset_manager and self.ml_registry:
            task_name = "ip.novelty"
            lambda_feat = self.feature_extractor.featurize_lambda_shells(
                (ligand_lambda or {}).get("descriptors", [])
            ) if self.feature_extractor else np.zeros(7)
            base_vec = np.array(
                [len(hits), self.context.enhancement_factor, len(report["generativeGraphCrosslinks"])],
                dtype=float,
            )
            feature_vec = (
                self.feature_extractor.combine_features(base_vec, lambda_feat)
                if self.feature_extractor
                else base_vec
            )
            if self.feature_extractor:
                feature_vec = self.feature_extractor.combine_features(feature_vec,
lambda_latent.flatten())
            label = 1.0 if report["noveltyAssessment"] == "High" else 0.5
            self.dataset_manager.register_record(
                task_name,
                feature_vec,
                label,
                {"ligandId": ligand_id, "source": "ip_agent"},
            )
            split = self.dataset_manager.build_split(task_name)
```

```
            model = self.ml_registry.get_model(task_name)
            if model is None and split.train_X.size:
                model = SimpleRegressor(f"{task_name}-reg", architecture="XGBoostProxy")
                metrics = model.train(split)
                self.ml_registry.register_model(task_name, model, metrics, split.metadata)
                training_record = lambda_shell_training_hook(
                    self.name,
                    self.context,
                    ligand_lambda or {"descriptors": self.context.lambda_shells},
                )
                self.observation_state.setdefault("trainingLogs", []).append(training_record)
            if model:
                normalization = split.normalization if split else {"mean": np.zeros_like(feature_vec),
"std": np.ones_like(feature_vec)}
                norm_vec = (
                    (feature_vec - normalization["mean"]) / normalization["std"]
                    if isinstance(normalization, dict)
                    else feature_vec
                )
                preds, uncert = model.predict_with_uncertainty(norm_vec.reshape(1, -1))
                ml_section = {
                    "model": model.describe(),
                    "noveltyScore": float(preds[0]),
                    "uncertainty": float(uncert[0]),
                    "dataset": split.metadata if split else {"records": 0},
                }
                report["mlAugmentation"] = ml_section
                if self.active_learning:
                    self.active_learning.evaluate_samples(
                        task_name,
                        [feature_vec],
                        preds,
                        uncert,
                        [{"ligandId": ligand_id}],
                    )
                if self.ml_api:
                    self.ml_api.register_endpoint("ip/novelty", task_name, model.version)
                    self.ml_api.log_call(
                        "ip/novelty",
                        {"ligandId": ligand_id},
                        {"noveltyScore": ml_section["noveltyScore"]},
                    )
        validated = self.validator.validate(self.name, report)
        await self.blackboard.post("ip", validated)
```

```
        return validated


class JobStatusAgent(AgentBase):
    def __init__(
        self,
        blackboard: QuantumBlackboard,
        context: QuantumContext,
        validator: PhysicalValidator,
        ml_registry: Optional[MLModelRegistry] = None,
        dataset_manager: Optional[DatasetManager] = None,
        feature_extractor: Optional[FeatureExtractor] = None,
        active_learning: Optional[ActiveLearningCoordinator] = None,
        ml_api: Optional[MLInferenceAPI] = None,
    ):
        super().__init__(
            "JobStatusAgent",
            blackboard,
            context,
            validator,
            ml_registry=ml_registry,
            dataset_manager=dataset_manager,
            feature_extractor=feature_extractor,
            active_learning=active_learning,
            ml_api=ml_api,
        )

    async def run(self) -> Dict[str, Any]:
        report = {
            "jobId": "qm-sim-job-1",
            "status": "COMPLETED",
            "resourceUtilization": {
                "simulationTime": "1800s",
                "quantumCircuitFidelity": float(np.clip(0.998 + 0.001 * random.random(), 0.0, 1.0)),
                "lambdaEnhancement": self.context.enhancement_factor,
            },
            "retrainMetrics": {
                "datasetsConsumed": 3,
                "lastRetrain": "scheduled",
            },
            "lambdaScaling": {
                "basis": self.context.lambda_basis,
                "shellDescriptors": self.context.lambda_shells,
            },
```

```
        }
        if self.dataset_manager and self.ml_registry:
            task_name = "operations.runtime"
            simulation_seconds = 1800.0
            feature_vec = np.array(
                [
                    simulation_seconds,
                    report["resourceUtilization"]["quantumCircuitFidelity"],
                    report["resourceUtilization"]["lambdaEnhancement"],
                    self.context.entanglement_entropy,
                ],
                dtype=float,
            )
            self.dataset_manager.register_record(
                task_name,
                feature_vec,
                simulation_seconds,
                {"jobId": report["jobId"], "source": "status_agent"},
            )
            split = self.dataset_manager.build_split(task_name)
            model = self.ml_registry.get_model(task_name)
            if model is None and split.train_X.size:
                model = SimpleRegressor(f"{task_name}-reg", architecture="TemporalCNNProxy")
                metrics = model.train(split)
                self.ml_registry.register_model(task_name, model, metrics, split.metadata)
                training_record = lambda_shell_training_hook(
                    self.name,
                    self.context,
                    {"descriptors": self.context.lambda_shells},
                )
                self.observation_state.setdefault("trainingLogs", []).append(training_record)
            if model:
                normalization = split.normalization if split else {"mean": np.zeros_like(feature_vec),
"std": np.ones_like(feature_vec)}
                norm_vec = (
                    (feature_vec - normalization["mean"]) / normalization["std"]
                    if isinstance(normalization, dict)
                    else feature_vec
                )
                preds, uncert = model.predict_with_uncertainty(norm_vec.reshape(1, -1))
                ml_section = {
                    "model": model.describe(),
                    "runtimeForecast": float(preds[0]),
                    "uncertainty": float(uncert[0]),
```

```
                "dataset": split.metadata if split else {"records": 0},
            }
            report["mlAugmentation"] = ml_section
            if self.active_learning:
                self.active_learning.evaluate_samples(
                    task_name,
                    [feature_vec],
                    preds,
                    uncert,
                    [{"jobId": report["jobId"]}],
                )
            if self.ml_api:
                self.ml_api.register_endpoint("operations/runtime", task_name, model.version)
                self.ml_api.log_call(
                    "operations/runtime",
                    {"jobId": report["jobId"]},
                    {"runtimeForecast": ml_section["runtimeForecast"]},
                )
        validated = self.validator.validate(self.name, report)
        await self.blackboard.post("status", validated)
        return validated


# ---------------------------------------------------------------------------
# Orchestration layer built on Golden Turing AI
# ---------------------------------------------------------------------------


class DrugDiscoverySimulation:
    def __init__(
        self,
        pdb_id: str,
        target_query: str,
        uniprot_accession: str,
        llm_model_path: Path,
        random_seed: int = DEFAULT_RANDOM_SEED,
    ) -> None:
        self.pdb_id = pdb_id
        self.target_query = target_query
        self.uniprot_accession = uniprot_accession
        self.llm_model_path = llm_model_path
        self.random_seed = random_seed
        set_global_random_seed(self.random_seed)
        self.data_client = PublicDataClient()
```

```
        self.geometry = GeometryParams()
        self.field = FieldParams()
        self.ent_params = EntanglementParams()
        self.quantum_engine = QuantumPhysicsEngine(self.geometry, self.field, self.ent_params)
        self.context = self.quantum_engine.compute_quantum_context()
        self.blackboard = QuantumBlackboard()
        self.llm = LightweightLLM(llm_model_path)
        self.GoldenTuringAI = load_golden_turing_ai()
        self.core_ai = self.GoldenTuringAI(config={"ai_state_dim": 64})
        self.core_ai.inject_blackboard_interface(self.blackboard)
        self.quantum_circuit_engine = QuantumCircuitEngine(self.context)
        self.memory_api = QuantumMemoryAPI()
        seed_ligands = [f"{self.target_query}-seed-{idx}" for idx in range(3)] + ["lig-novel-001"]
        try:
            self.quantum_reference =
self.quantum_circuit_engine.generate_reference_dataset(seed_ligands)
        except Exception as exc:  # pragma: no cover - defensive fallback
            self.quantum_reference = {
                "samples": [],
                "statistics": {"energyRange": {"min": -40.0, "max": -2.0}, "meanEnergy": -12.0,
"meanEntropy": 0.4},
                "error": str(exc),
            }
        self.validator = PhysicalValidator(self.context, self.quantum_reference)
        self.feature_extractor = FeatureExtractor()
        self.dataset_manager = DatasetManager(
            self.feature_extractor,
            random_seed=self.random_seed,
        )
        self.ml_registry = MLModelRegistry()
        self.active_learning = ActiveLearningCoordinator()
        self.ml_api = MLInferenceAPI()
        self.validation_engine = StatisticalValidationEngine(random_seed=self.random_seed)
        self.explainability = ExplainabilityEngine(random_seed=self.random_seed)
        self.baseline_suite = BaselineModelSuite(
            self.validation_engine,
            self.explainability,
            random_seed=self.random_seed,
        )
        self.hyperparameter_optimizer = HyperparameterOptimizer(
            self.validation_engine,
            random_seed=self.random_seed,
        )
        self.visualizer = TrainingVisualizationLogger()
```

```
        self.cross_validation_results: Dict[str, Any] = {}
        self.benchmark_summary: Dict[str, Any] = {}
        self.baseline_results: Dict[str, Any] = {}
        self.hyperparameter_results: Dict[str, Any] = {}
        self.visualization_artifacts: List[str] = []
        self.gtai_adapter = GoldenTuringDDSAdapter(
            self.core_ai,
            self.blackboard,
            self.validator,
            self.quantum_reference,
            memory_api=self.memory_api,
        )
        self._bootstrap_ml_datasets()
        self.orchestration_directives = {
            "physicalRealism": "Enhance physical realism and chemical accuracy using quantum
chemistry and ensemble modeling",
            "additionalFunctions": "Extend agents to ligand library generation, combinatorial
synthesis, pharmacophore mining, IP audits",
            "quantumData": "Integrate quantum circuit engine outputs for training and constraint
enforcement",
        }

    def _bootstrap_ml_datasets(self) -> None:
        samples = self.quantum_reference.get("samples", [])
        if samples:
            stats = self.quantum_reference.get("statistics", {})
            mean_energy = stats.get("meanEnergy", -8.0)
            for sample in samples:
                quantum_feat = self.feature_extractor.featurize_quantum_sample(sample)
                smiles_feat = self.feature_extractor.featurize_smiles(sample.get("ligandId", ""))
                lambda_feat =
self.feature_extractor.featurize_lambda_shells(self.context.lambda_shells)
                combined_feat = self.feature_extractor.combine_features(quantum_feat, smiles_feat,
lambda_feat)
                self.dataset_manager.register_record(
                    "quantum.bindingEnergy",
                    combined_feat,
                    sample.get("bindingEnergy", -8.0),
                    {"ligandId": sample.get("ligandId"), "source": "bootstrap"},
                )
                label = 1.0 if sample.get("bindingEnergy", 0.0) < mean_energy else 0.0
                self.dataset_manager.register_record(
                    "quantum.highAffinity",
                    combined_feat,
```

```
                label,
                {"ligandId": sample.get("ligandId"), "source": "bootstrap"},
            )
            lambda_latent = np.asarray(
                LambdaScalingToolkit.fingerprint(self.context.lambda_shells),
                dtype=float,
            )
            ligand_feat = self.feature_extractor.combine_features(smiles_feat, lambda_feat)
            ligand_feat = self.feature_extractor.combine_features(ligand_feat, lambda_latent)
            self.dataset_manager.register_record(
                "ligand.bindingAffinity",
                ligand_feat,
                sample.get("bindingEnergy", -8.0),
                {"ligandId": sample.get("ligandId"), "source": "bootstrap"},
            )
        binding_split = self.dataset_manager.build_split("quantum.bindingEnergy")
        if binding_split.train_X.size:
            binding_model = GraphSurrogateModel("quantum.bindingEnergy-bootstrap")
            metrics = binding_model.train(binding_split)
            self.ml_registry.register_model("quantum.bindingEnergy", binding_model, metrics,
binding_split.metadata)
            self.ml_api.register_endpoint("bootstrap/quantumBinding", "quantum.bindingEnergy",
binding_model.version)
            self.validation_engine.log_evaluation("bootstrap.quantum.bindingEnergy", metrics)
            step = len(self.visualizer.metric_history["bootstrap.quantum.bindingEnergy"])
            self.visualizer.log_metrics("bootstrap.quantum.bindingEnergy", step, metrics)
        high_split = self.dataset_manager.build_split("quantum.highAffinity")
        if high_split.train_X.size:
            high_classifier = SimpleClassifier("quantum.highAffinity-bootstrap",
architecture="EnsembleProxy")
            metrics = high_classifier.train(high_split)
            self.ml_registry.register_model("quantum.highAffinity", high_classifier, metrics,
high_split.metadata)
            self.ml_api.register_endpoint("bootstrap/highAffinity", "quantum.highAffinity",
high_classifier.version)
            self.validation_engine.log_evaluation("bootstrap.quantum.highAffinity", metrics)
            step = len(self.visualizer.metric_history["bootstrap.quantum.highAffinity"])
            self.visualizer.log_metrics("bootstrap.quantum.highAffinity", step, metrics)
    # Seed safety dataset with conservative priors
    for idx in range(3):
        base_vec = np.array(
            [
                0.2 + 0.1 * idx,
                0.55 + 0.05 * idx,
```

```
                4.0 + idx,
                0.1 + 0.02 * idx,
                0.0,
                0.0,
            ],
            dtype=float,
        )
        lambda_feat =
self.feature_extractor.featurize_lambda_shells(self.context.lambda_shells)
        lambda_latent = np.asarray(
            LambdaScalingToolkit.fingerprint(self.context.lambda_shells),
            dtype=float,
        )
        feature_vec = self.feature_extractor.combine_features(base_vec, lambda_feat)
        feature_vec = self.feature_extractor.combine_features(feature_vec, lambda_latent)
        self.dataset_manager.register_record(
            "safety.toxicity",
            feature_vec,
            0.0,
            {"ligandId": f"seed-{idx}", "source": "bootstrap"},
        )
    safety_split = self.dataset_manager.build_split("safety.toxicity")
    if safety_split.train_X.size:
        metrics = {"mae": 0.0, "rmse": 0.0, "r2": 0.0, "bhattacharyya": 0.0}
        self.validation_engine.log_evaluation("bootstrap.safety.toxicity", metrics)
        step = len(self.visualizer.metric_history["bootstrap.safety.toxicity"])
        self.visualizer.log_metrics("bootstrap.safety.toxicity", step, metrics)

    benchmark_names = ("DUD-E", "ChEMBL", "ZINC15")
    self.benchmark_summary =
self.dataset_manager.prepare_benchmarks(benchmark_names, self.context, limit=200)

    for task in ("quantum.highAffinity", "safety.toxicity"):
        records = self.dataset_manager.records.get(task, [])
        if not records:
            continue
        features = np.stack([entry["features"] for entry in records])
        labels = np.array([entry["label"] for entry in records], dtype=float)
        task_type = self.dataset_manager.get_task_type(task)
        if task_type == "classification" and RandomForestClassifier is not None:
            factory: Callable[[], Any] = lambda: RandomForestClassifier(n_estimators=64,
random_state=self.random_seed)
        elif task_type == "regression" and RandomForestRegressor is not None:
```

```
            factory = lambda: RandomForestRegressor(n_estimators=64,
random_state=self.random_seed)
        else:
            factory = lambda: GraphSurrogateModel(f"kfold-{task}")
        result = self.validation_engine.k_fold_cross_validation(factory, features, labels, folds=5,
task_type=task_type)
        self.cross_validation_results[task] = result
        aggregate = result.get("aggregate", {})
        if aggregate:
            self.validation_engine.log_evaluation(f"kfold.{task}", aggregate)
            step = len(self.visualizer.metric_history[f"kfold.{task}"])
            self.visualizer.log_metrics(f"kfold.{task}", step, aggregate)

    self.baseline_results = self.baseline_suite.train_and_evaluate(self.dataset_manager)
    for task, details in self.baseline_results.items():
        comparison = details.get("comparison", {})
        best = {metric: spec.get("score") for metric, spec in comparison.get("best", {}).items() if
isinstance(spec, dict)}
        if best:
            self.validation_engine.log_evaluation(f"baseline.{task}", best)
            step = len(self.visualizer.metric_history[f"baseline.{task}"])
            self.visualizer.log_metrics(f"baseline.{task}", step, best)

    self.hyperparameter_results = {}
    for dataset_name, meta in self.benchmark_summary.items():
        task = meta.get("task")
        if not task:
            continue
        split = self.dataset_manager.get_split(task)
        if not split or not split.train_X.size:
            continue
        task_type = self.dataset_manager.get_task_type(task)
        results: Dict[str, Any] = {}
        if task_type == "classification" and RandomForestClassifier is not None:
            results["grid"] = self.hyperparameter_optimizer.grid_search(
                lambda params: RandomForestClassifier(
                    n_estimators=int(params.get("n_estimators", 128)),
                    max_depth=None if params.get("max_depth") in (None, "None") else
int(params.get("max_depth")),
                    random_state=self.random_seed,
                ),
                {"n_estimators": [64, 128, 256], "max_depth": [None, 8, 12]},
                split,
                task_type,
```

```
            )
            results["bayesian"] = self.hyperparameter_optimizer.bayesian_search(
                lambda params: RandomForestClassifier(
                    n_estimators=max(32, int(params.get("n_estimators", 100))),
                    max_depth=int(max(4, params.get("max_depth", 10))),
                    random_state=self.random_seed,
                ),
                {"n_estimators": (32, 256), "max_depth": (4, 16)},
                split,
                task_type,
                iterations=10,
            )
        elif task_type != "classification" and RandomForestRegressor is not None:
            results["grid"] = self.hyperparameter_optimizer.grid_search(
                lambda params: RandomForestRegressor(
                    n_estimators=int(params.get("n_estimators", 128)),
                    max_depth=None if params.get("max_depth") in (None, "None") else
int(params.get("max_depth")),
                    random_state=self.random_seed,
                ),
                {"n_estimators": [64, 128, 256], "max_depth": [None, 8, 14]},
                split,
                task_type,
            )
            results["bayesian"] = self.hyperparameter_optimizer.bayesian_search(
                lambda params: RandomForestRegressor(
                    n_estimators=max(32, int(params.get("n_estimators", 100))),
                    max_depth=int(max(4, params.get("max_depth", 12))),
                    random_state=self.random_seed,
                ),
                {"n_estimators": (32, 256), "max_depth": (4, 20)},
                split,
                task_type,
                iterations=10,
            )
        else:
            results["grid"] = None
            results["bayesian"] = None
            self.hyperparameter_optimizer.records.append(
                {
                    "method": "unavailable",
                    "params": {},
                    "metrics": {},
                    "task": task,
```

```
                    "reason": "Baseline library missing for task type",
                }
            )
        self.hyperparameter_results[task] = results
        best_metrics = {}
        for strategy in ("grid", "bayesian"):
            metrics = (results.get(strategy) or {}).get("bestMetrics", {})
            for key, value in metrics.items():
                if isinstance(value, (int, float)) and not math.isnan(value):
                    best_metrics[f"{strategy}_{key}"] = float(value)
        if best_metrics:
            self.validation_engine.log_evaluation(f"hyperopt.{task}", best_metrics)
            step = len(self.visualizer.metric_history[f"hyperopt.{task}"])
            self.visualizer.log_metrics(f"hyperopt.{task}", step, best_metrics)

    self.visualization_artifacts = self.visualizer.render()

async def _instantiate_agents(self) -> List[AgentBase]:
    uniprot_meta = self.data_client.fetch_uniprot_metadata(self.uniprot_accession)
    agents: List[AgentBase] = [
        StructuralAnalysisAgent(
            self.blackboard,
            self.context,
            self.validator,
            self.data_client,
            self.pdb_id,
            ml_registry=self.ml_registry,
            dataset_manager=self.dataset_manager,
            feature_extractor=self.feature_extractor,
            active_learning=self.active_learning,
            ml_api=self.ml_api,
        ),
        LigandDiscoveryAgent(
            self.blackboard,
            self.context,
            self.validator,
            self.data_client,
            self.llm,
            self.target_query,
            self.quantum_reference,
            ml_registry=self.ml_registry,
            dataset_manager=self.dataset_manager,
            feature_extractor=self.feature_extractor,
            active_learning=self.active_learning,
```

```
        ml_api=self.ml_api,
    ),
    QuantumSimulationAgent(
        self.blackboard,
        self.context,
        self.validator,
        uniprot_meta,
        self.quantum_reference,
        ml_registry=self.ml_registry,
        dataset_manager=self.dataset_manager,
        feature_extractor=self.feature_extractor,
        active_learning=self.active_learning,
        ml_api=self.ml_api,
    ),
    SynthesisPlannerAgent(
        self.blackboard,
        self.context,
        self.validator,
        self.llm,
        ml_registry=self.ml_registry,
        dataset_manager=self.dataset_manager,
        feature_extractor=self.feature_extractor,
        active_learning=self.active_learning,
        ml_api=self.ml_api,
    ),
    ScreeningAgent(
        self.blackboard,
        self.context,
        self.validator,
        self.quantum_reference,
        ml_registry=self.ml_registry,
        dataset_manager=self.dataset_manager,
        feature_extractor=self.feature_extractor,
        active_learning=self.active_learning,
        ml_api=self.ml_api,
    ),
    SafetyAgent(
        self.blackboard,
        self.context,
        self.validator,
        self.quantum_reference,
        ml_registry=self.ml_registry,
        dataset_manager=self.dataset_manager,
        feature_extractor=self.feature_extractor,
```

```
            active_learning=self.active_learning,
            ml_api=self.ml_api,
        ),
        IPAgent(
            self.blackboard,
            self.context,
            self.validator,
            self.data_client,
            self.target_query,
            ml_registry=self.ml_registry,
            dataset_manager=self.dataset_manager,
            feature_extractor=self.feature_extractor,
            active_learning=self.active_learning,
            ml_api=self.ml_api,
        ),
        JobStatusAgent(
            self.blackboard,
            self.context,
            self.validator,
            ml_registry=self.ml_registry,
            dataset_manager=self.dataset_manager,
            feature_extractor=self.feature_extractor,
            active_learning=self.active_learning,
            ml_api=self.ml_api,
        ),
    ]
    for agent in agents:
        self.core_ai.register_agent(agent.name, agent)
        self.gtai_adapter.register_agent(agent)
    return agents

def _sanitize_for_json(self, value: Any) -> Any:
    if isinstance(value, (str, int, float, bool)) or value is None:
        return value
    if isinstance(value, np.ndarray):
        return value.tolist()
    if isinstance(value, dict):
        return {str(key): self._sanitize_for_json(val) for key, val in value.items()}
    if isinstance(value, (list, tuple)):
        return [self._sanitize_for_json(item) for item in value]
    if isinstance(value, set):
        return [self._sanitize_for_json(item) for item in value]
    if hasattr(value, "to_dict") and callable(getattr(value, "to_dict")):
        return self._sanitize_for_json(value.to_dict())
```

```
    return repr(value)

def _extract_shell_snapshot(self, context: QuantumContext) -> Dict[str, Any]:
    stats = QuantumContext.shell_statistics(context.lambda_shells)
    return {
        "shellCount": stats["shellCount"],
        "entropyMean": stats["entropyMean"],
        "entropyStd": stats["entropyStd"],
        "bhattacharyyaMean": stats["bhattacharyyaMean"],
        "curvatureMean": stats["curvatureMean"],
        "entropyDistribution": stats["entropyDistribution"],
        "occupancyDistribution": stats["occupancyDistribution"],
    }

def _summarize_agent_output(self, agent_output: Any) -> Dict[str, Any]:
    if not isinstance(agent_output, dict):
        return {"value": self._sanitize_for_json(agent_output)}
    summary: Dict[str, Any] = {
        "reportId": agent_output.get("reportId"),
        "keys": sorted(agent_output.keys()),
    }
    list_counts: Dict[str, int] = {}
    dict_keys: Dict[str, List[str]] = {}
    for key, value in agent_output.items():
        if isinstance(value, list):
            list_counts[key] = len(value)
        elif isinstance(value, dict):
            dict_keys[key] = sorted(value.keys())
    if list_counts:
        summary["listCounts"] = list_counts
    if dict_keys:
        summary["dictKeys"] = dict_keys
    return summary

def compute_step_metrics(
    self,
    context: QuantumContext,
    previous_context: Optional[QuantumContext],
) -> Dict[str, Any]:
    metrics = {
        "entropicFidelity": None,
        "shellEntropyDelta": None,
        "lambdaShellStability": None,
    }
```

```
        if previous_context is None:
            return metrics
        current_stats = QuantumContext.shell_statistics(context.lambda_shells)
        previous_stats = QuantumContext.shell_statistics(previous_context.lambda_shells)
        if current_stats["shellCount"] == 0 or previous_stats["shellCount"] == 0:
            return metrics

        def _pad(values: List[float], length: int) -> np.ndarray:
            arr = np.asarray(values, dtype=float)
            if arr.size < length:
                arr = np.pad(arr, (0, length - arr.size), constant_values=0.0)
            return arr

        max_len = max(
            len(current_stats["entropyDistribution"]),
            len(previous_stats["entropyDistribution"]),
        )
        curr_entropy = _pad(current_stats["entropyDistribution"], max_len)
        prev_entropy = _pad(previous_stats["entropyDistribution"], max_len)
        max_occ_len = max(
            len(current_stats["occupancyDistribution"]),
            len(previous_stats["occupancyDistribution"]),
        )
        curr_occ = _pad(current_stats["occupancyDistribution"], max_occ_len)
        prev_occ = _pad(previous_stats["occupancyDistribution"], max_occ_len)
        fidelity = float(np.clip(np.sum(np.sqrt(curr_entropy * prev_entropy)), 0.0, 1.0))
        stability = float(np.clip(np.sum(np.sqrt(curr_occ * prev_occ)), 0.0, 1.0))
        metrics.update(
            {
                "entropicFidelity": fidelity,
                "shellEntropyDelta": float(current_stats["entropyMean"] -
previous_stats["entropyMean"]),
                "lambdaShellStability": stability,
            }
        )
        return metrics

    def _record_agent_step(
        self,
        agent_name: str,
        agent_output: Dict[str, Any],
        before_snapshot: Dict[str, Any],
        blackboard_channels_before: Sequence[str],
    ) -> None:
```

```
    previous_context = QuantumContext.from_snapshot(before_snapshot)
    step_metrics = self.compute_step_metrics(self.context, previous_context)
    shell_entry = self._extract_shell_snapshot(self.context)
    shell_entry.update({
        "stage": agent_name,
        "timestamp": time.time(),
        "stepMetrics": step_metrics,
    })
    self.shell_log.append(shell_entry)
    after_snapshot = self.context.to_snapshot()
    context_diff = QuantumContext.diff_snapshots(before_snapshot, after_snapshot)
    trace_entry = {
        "timestamp": time.time(),
        "agent": agent_name,
        "inputs": {
            "contextSummary": self._extract_shell_snapshot(previous_context),
            "blackboardChannels": list(blackboard_channels_before),
        },
        "outputs": self._summarize_agent_output(agent_output),
        "blackboardChannelsAfter": list(self.blackboard.posts.keys()),
        "stepMetrics": step_metrics,
        "contextDiff": context_diff,
    }
    self.agent_traces.append(trace_entry)
    with self.agent_trace_path.open("a", encoding="utf-8") as handle:
        handle.write(json.dumps(self._sanitize_for_json(trace_entry)) + "\n")

async def run(self) -> Dict[str, Any]:
    run_start = time.time()
    output_dir = Path("outputs")
    output_dir.mkdir(parents=True, exist_ok=True)
    self.shell_log: List[Dict[str, Any]] = []
    self.agent_traces: List[Dict[str, Any]] = []
    self.agent_trace_path = output_dir / "agent_trace.log"
    self.agent_trace_path.write_text("")
    agents = await self._instantiate_agents()
    reports: Dict[str, Any] = {}
    await self.blackboard.post(
        "quantumTrainingData",
        self.validator.validate(
            "QuantumCircuitEngine",
            {
                "reportId": "quantum-training-reference",
                "directive": self.orchestration_directives["quantumData"],
```

```
                "dataset": self.quantum_reference,
            },
        ),
    )
    ml_status_payload = {
        "reportId": "ml-status-initial",
        "models": self.ml_registry.describe(),
        "datasets": {task: len(records) for task, records in self.dataset_manager.records.items()},
        "activeLearning": self.active_learning.describe(),
        "api": self.ml_api.describe(),
    }
    await self.blackboard.post(
        "mlStatus",
        self.validator.validate("MLOrchestrator", ml_status_payload),
    )
    # run structural first to seed pockets
    self.gtai_adapter.before_agent_run(agents[0].name)
    structural_before_snapshot = self.context.to_snapshot()
    structural_blackboard_before = list(self.blackboard.posts.keys())
    structural_report = await agents[0].run()
    structural_report, structural_meta = self.gtai_adapter.after_agent_report(
        agents[0].name, structural_report
    )
    if structural_meta:
        structural_report["gtaiIntegration"] = structural_meta
    reports["structural"] = structural_report
    self._record_agent_step(
        agents[0].name,
        structural_report,
        structural_before_snapshot,
        structural_blackboard_before,
    )
    # run discovery + quantum sequentially to respect dependencies
    for agent in agents[1:3]:
        self.gtai_adapter.before_agent_run(agent.name)
        before_snapshot = self.context.to_snapshot()
        blackboard_before = list(self.blackboard.posts.keys())
        agent_report = await agent.run()
        processed, integration_meta = self.gtai_adapter.after_agent_report(agent.name,
agent_report)
        if integration_meta:
            processed["gtaiIntegration"] = integration_meta
        reports[agent.name] = processed
        self._record_agent_step(agent.name, processed, before_snapshot, blackboard_before)
```

```
        # run remaining agents sequentially for detailed logging
        for agent in agents[3:]:
            self.gtai_adapter.before_agent_run(agent.name)
            before_snapshot = self.context.to_snapshot()
            blackboard_before = list(self.blackboard.posts.keys())
            agent_report = await agent.run()
            processed, integration_meta = self.gtai_adapter.after_agent_report(agent.name,
agent_report)
            if integration_meta:
                processed["gtaiIntegration"] = integration_meta
            reports[agent.name] = processed
            self._record_agent_step(agent.name, processed, before_snapshot, blackboard_before)
        simulation_reflection = self.gtai_adapter.run_recursive_simulation()
        avg_reward = statistics.mean(self.gtai_adapter.awareness_changes) if
self.gtai_adapter.awareness_changes else 0.0
        tuning_meta = self.gtai_adapter.tune_analysis_parameters(avg_reward)
        memory_audit = self.gtai_adapter.analyze_memory()
        reports["directives"] = self.orchestration_directives
        await self.gtai_adapter.flush_blackboard()
        snapshot = await self.blackboard.snapshot()
        reports["blackboard"] = snapshot
        reports["mlStatus"] = {
            "models": self.ml_registry.describe(),
            "datasets": {task: len(records) for task, records in self.dataset_manager.records.items()},
            "activeLearning": self.active_learning.describe(),
            "api": self.ml_api.describe(),
        }
        gtai_summary = self.gtai_adapter.compile_summary()
        gtai_summary["latestReflection"] = simulation_reflection
        gtai_summary["latestTuning"] = tuning_meta
        gtai_summary["latestMemoryAudit"] = memory_audit
        reports["gtaiIntegration"] = gtai_summary
        reports["benchmarks"] = self.benchmark_summary
        reports["baselineComparisons"] = self.baseline_results
        reports["hyperparameterOptimization"] = self.hyperparameter_results
        reports["hyperparameterSearchLog"] = self.hyperparameter_optimizer.records
        reports["crossValidation"] = self.cross_validation_results
        reports["visualizations"] = self.visualization_artifacts
        reports["validationHistory"] = self.validation_engine.history
        shell_trace_path = output_dir / "shell_trace.json"
        shell_trace_path.write_text(json.dumps(self._sanitize_for_json(self.shell_log), indent=2))
        timestamp = datetime.utcnow()
        ligand_predictions = (
            reports.get("LigandDiscoveryAgent", {})
```

```
            .get("mlAugmentation", {})
            .get("rankedCandidates", [])
        )
        screening_predictions = (
            reports.get("ScreeningAgent", {})
            .get("mlAugmentation", {})
            .get("predictions", [])
        )
        safety_assessment = reports.get("SafetyAgent", {}).get("mlAugmentation", {})
        agent_trace_summaries = [
            {
                "agent": entry.get("agent"),
                "timestamp": entry.get("timestamp"),
                "stepMetrics": entry.get("stepMetrics"),
                "contextDiff": entry.get("contextDiff"),
            }
            for entry in self.agent_traces
        ]
        shell_log_summary = {
            "entries": len(self.shell_log),
            "stages": [entry.get("stage") for entry in self.shell_log],
            "entropyMeanTrajectory": [entry.get("entropyMean") for entry in self.shell_log],
            "bhattacharyyaTrajectory": [entry.get("bhattacharyyaMean") for entry in self.shell_log],
        }
        final_shell_state = self.shell_log[-1] if self.shell_log else None
        summary_payload = {
            "generatedAt": timestamp.isoformat() + "Z",
            "runtimeSeconds": float(time.time() - run_start),
            "finalShellState": final_shell_state,
            "shellLogSummary": shell_log_summary,
            "agentTraceSummaries": agent_trace_summaries,
            "predictions": {
                "topLigandCandidates": ligand_predictions[:5],
                "screeningPredictions": screening_predictions[:5],
                "safetyAssessment": safety_assessment,
            },
        }
        summary_path = output_dir /
f"simulation_metrics_{timestamp.strftime('%Y%m%dT%H%M%SZ')}.json"
        summary_path.write_text(json.dumps(self._sanitize_for_json(summary_payload),
indent=2))
        reports["instrumentation"] = {
            "shellTrace": str(shell_trace_path),
            "agentTrace": str(self.agent_trace_path),
```

```
            "summary": str(summary_path),
        }
        return reports


# ---------------------------------------------------------------------------
# CLI entrypoint
# ---------------------------------------------------------------------------


def main(argv: Optional[Iterable[str]] = None) -> None:
    parser = argparse.ArgumentParser(description="Run the Golden Turing drug discovery
simulation")
    parser.add_argument("--pdb-id", default="4AKE", help="PDB identifier for structural analysis")
    parser.add_argument("--target-query", default="aspirin", help="Ligand design query
keyword")
    parser.add_argument("--uniprot", default="P35354", help="UniProt accession for target
metadata")
    parser.add_argument(
        "--llm-model-path",
        default=os.path.join(os.getcwd(), "models", "tinyllama-1.1b-chat-v1.0.Q4_K_M.gguf"),
        help="Path to the TinyLlama GGUF model",
    )
    parser.add_argument(
        "--random-seed",
        type=int,
        default=DEFAULT_RANDOM_SEED,
        help="Random seed for reproducible agent initialization",
    )
    args = parser.parse_args(list(argv) if argv is not None else None)

    simulation = DrugDiscoverySimulation(
        pdb_id=args.pdb_id,
        target_query=args.target_query,
        uniprot_accession=args.uniprot,
        llm_model_path=Path(args.llm_model_path),
        random_seed=args.random_seed,
    )
    reports = asyncio.run(simulation.run())
    print(json.dumps(reports, indent=2))
    output_dir = Path("outputs")
    output_dir.mkdir(parents=True, exist_ok=True)
    output_file = output_dir / "latest_drug_discovery_simulation.json"
    output_file.write_text(json.dumps(reports, indent=2))
```

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