The Refined Refined

```
import os
import json
import logging
import time
import random
import numpy as np
import networkx as nx
import scipy.sparse as sp
import qutip as qt
import plotly.graph_objects as go
from datetime import datetime
from collections import deque
# The following imports are used in the QEC section (ldpc, stim) but are
# often external packages and may require separate installation.
try:
    from ldpc import bposd_decoder
    from ldpc.codes import tanner_code
except ImportError:
    # Placeholder classes/functions if ldpc is not installed
    bposd_decoder = object
    tanner_code = lambda *args, **kwargs: (None, None, None)

try:
    import stim
except ImportError:
    # Placeholder class/function if stim is not installed
    stim = object

#
================================================================================
====
# Global Placeholders/Stubs (Required by Persistence and Run Functions)
#
================================================================================
====
```

```
# Global Memory Store (used by persistence functions)
REFINEMENT_MEMORY = deque(maxlen=200)

# Placeholder for the main network graph
G = nx×Graph()

# Placeholder for the external function that drives refinement cycles.
# It should return a dictionary structure that 'run_and_checkpoint' expects.
def run_refinement_cycles(query, params, G, cycles=6, memory_window=5,
allow_structure_change=False):
    logging.warning("Stub: run_refinement_cycles called. Returning placeholder data.")
    # Simulate a single cycle for simplicity
    summary = [{
        "cycle": 1,
        "coherence": random.uniform(0.5, 0.9),
        "chaos": random.uniform(0.1, 0.5),
        "vibration_before": params.get('vibration_freq', 0.5),
        "vibration_after": params.get('vibration_freq', 0.5) * 1.3,
        "distortions": ["Example Distortion"],
        "timestamp": time.time()
    }]
    # Simulate an update
    params['vibration_freq'] *= 1.3
    return {"summary": summary, "memory": list(REFINEMENT_MEMORY) + summary,
"final_params": params}

#
================================================================================
====
# Configuration and Logging
#
================================================================================
====

# Configure logging
logging.basicConfig(filename='simulation.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')
```

```
# Configuration dictionary
SYSTEM_CONFIG = {
    "oxygen_range": (0.7, 1.0),
    "ether_range": (0.5, 0.9),
    "vibration_range": (0.3, 0.8),
    "light_range": (0.6, 1.0),
    "muscle_range": (0.4, 0.9),
    "tendon_range": (0.5, 0.95),
    "ligament_range": (0.6, 1.0),
    "cartilage_range": (0.5, 0.95),
    "synovial_range": (0.6, 1.0),
    "bursa_range": (0.5, 0.95),
    "fascia_range": (0.6, 1.0),
    "nervous_range": (0.5, 0.95),
    "endocrine_range": (0.6, 1.0),
    "circulatory_range": (0.7, 1.0),
    "respiratory_range": (0.6, 1.0),
    "immune_range": (0.5, 0.95),
    "lymphatic_range": (0.6, 1.0),
    "digestive_range": (0.5, 0.95),
    "urinary_range": (0.6, 1.0),
    "reproductive_range": (0.5, 0.95),
    "integumentary_range": (0.6, 1.0),
    "musculoskeletal_range": (0.7, 1.0),
    "sensory_range": (0.6, 1.0),
    "cardiovascular_range": (0.7, 1.0),
    "thermoregulatory_range": (0.6, 1.0),
    "excretory_range": (0.6, 1.0),
    "perception_range": (0.6, 1.0),
    "cognition_range": (0.7, 1.0),
    "integrity_range": (0.8, 1.0),
    "sapience_range": (0.7, 1.0),
    "excretion_ranges": {
        "integumentary_excretion": (0.5, 0.95),
        "respiratory_excretion": (0.6, 1.0),
        "nervous_excretion": (0.5, 0.95),
        "endocrine_extraction": (0.6, 1.0),
```

```
    "immune_excretion": (0.5, 0.95),
    "digestive_excretion": (0.6, 1.0),
    "reproductive_excretion": (0.5, 0.95),
    "sensory_excretion": (0.6, 1.0),
    "cognitive_excretion": (0.6, 1.0),
    "emotional_excretion": (0.6, 1.0),
    "spiritual_excretion": (0.6, 1.0),
    "energetic_excretion": (0.6, 1.0),
    "quantum_excretion": (0.6, 1.0),
    "vibrational_excretion": (0.6, 1.0)
},
"thresholds": {
    "clustering": 0.25,
    "emotrix_score": 0.7,
    "level_threshold": 0.2,
    "rate_thresholds": {
        "muscle_tension": 0.6,
        "tendon_elasticity": 0.7,
        "ligament_strength": 0.8,
        "cartilage_resilience": 0.7,
        "synovial_viscosity": 0.8,
        "bursa_cushioning": 0.7,
        "fascia_connectivity": 0.8,
        "nervous_sensitivity": 0.7,
        "endocrine_balance": 0.8,
        "circulatory_flow": 0.9,
        "respiratory_capacity": 0.8,
        "immune_strength": 0.7,
        "lymphatic_flow": 0.8,
        "digestive_efficiency": 0.7,
        "urinary_filtration": 0.8,
        "reproductive_vitality": 0.7,
        "integumentary_resilience": 0.8,
        "musculoskeletal_strength": 0.9,
        "sensory_acuity": 0.8,
        "cardiovascular_rhythm": 0.9,
        "thermoregulatory_balance": 0.8,
        "excretory_clearance": 0.8,
```

```
        "default_excretion": 0.8,
        "integumentary_excretion": 0.7,
        "nervous_excretion": 0.7,
        "immune_excretion": 0.7,
        "reproductive_excretion": 0.7,
        "perception_alignment": 0.8,
        "cognition_coherence": 0.85,
        "integrity_steadfastness": 0.9,
        "sapience_depth": 0.85,
        "default": 0.8
    }
  }
}

#
================================================================================
====
# Persistence Utilities
#
================================================================================
====

def save_refinement_memory(path="grok_refinement_memory.json", memory=None):
  """Save REFINEMENT_MEMORY (or provided memory) to disk as JSON."""
  try:
    mem = memory if memory is not None else REFINEMENT_MEMORY
    os.makedirs(os.path.dirname(path) or ".", exist_ok=True)
    # backup existing
    if os.path.exists(path):
      stamp = datetime×utcnow()×strftime("%Y%m%dT%H%M%SZ")
      backup = f"{path}.bak.{stamp}"
      with open(path, "r", encoding="utf-8") as f:
        old = f×read()
      with open(backup, "w", encoding="utf-8") as f:
        f.write(old)
    # write new
    with open(path, "w", encoding="utf-8") as f:
      json.dump({"saved_at": datetime.utcnow().isoformat(), "memory": mem}, f, default=str,
```

```
indent=2)
        logging.info("Saved refinement memory to %s", path)
        return path
    except Exception as e:
        logging.exception("Failed to save refinement memory: %s", e)
        return None

def load_refinement_memory(path="grok_refinement_memory.json"):
    """Load memory from disk. Returns list (may be empty) and populates
REFINEMENT_MEMORY."""
    try:
        if not os.path.exists(path):
            logging.info("No memory file found at %s; returning empty memory.", path)
            return []
        with open(path, "r", encoding="utf-8") as f:
            data = json×load(f)
        mem = data×get("memory", [])
        # shallow replace of module memory
        REFINEMENT_MEMORY.clear()
        REFINEMENT_MEMORY.extend(mem)
        logging.info("Loaded %d memory snapshots from %s", len(mem), path)
        return REFINEMENT_MEMORY
    except Exception as e:
        logging.exception("Failed to load refinement memory: %s", e)
        return []

#
================================================================================
====
# Narrative Builders
#
================================================================================
====

def build_poetic_narrative(summaries, voice="soft"):
    """Convert the numeric summaries into a poetic + clinical narrative."""
    paragraphs = []
    for s in summaries:
```

```
        c = s×get("coherence")
        ch = s×get("chaos")
        vf_b = s×get("vibration_before") or 0.0
        vf_a = s×get("vibration_after") or 0.0
        distort = s×get("distortions") or []
        tstamp = datetime×utcfromtimestamp(s×get("timestamp", time.time())).strftime("%Y-%m-%d
%H:%M:%SZ")
        if voice == "clinical":
            p = f"[{tstamp}] Cycle {s['cycle']}: coherence={c:.3f} chaos={ch:.3f} | vf {vf_b:.3f}->{vf_a:.3f}
| distortions={len(distort)}"
        elif voice == "mixed":
            p = f"{tstamp} - Cycle {s['cycle']}: coherence={c:.3f}, chaos={ch:.3f}. Vibration {vf_b:.3f}
→{vf_a:.3f}. Distortions: {len(distort)}."
        else: # soft/poetic
            mood = "steadied" if (c is not None and c > (ch or 0)) else "restless"
            p = (f"{tstamp} - Cycle {s['cycle']}: the weave feels {mood}; "
                f"a pulse of {c:.3f} coherence against {ch:.3f} chaos. "
                f"Vibration shifted {vf_b:.3f} -> {vf_a:.3f}. "
                f"Small echoes: {len(distort)} distortions noticed.")
        paragraphs.append(p)
    return "\n".join(paragraphs)

def sparkline(series, length=10):
    """Creates a tiny ascii sparkline-like trend visualization."""
    if not series:
        return ""
    mn, mx = min(series), max(series)
    if mx - mn < 1e-6:
        return " " * len(series)
    # Corrected set of ticks
    ticks = " ▁▂▃▄▅▆▇█"
    res = ""
    for v in series:
        idx = int((v - mn) / (mx - mn) × (len(ticks) - 1))
        res += ticks[idx]
    return res

def textual_dashboard(summaries, last_n=6):
```

```
    """Returns a short textual dashboard: trend lines and highlights."""
    out = []
    recent = summaries[-last_n:]
    coherence_series = [s.get("coherence") or 0.0 for s in recent]
    chaos_series = [s.get("chaos") or 0.0 for s in recent]

    avg_coh = sum(coherence_series) / len(coherence_series) if coherence_series else 0.0
    avg_chaos = sum(chaos_series) / len(chaos_series) if chaos_series else 0.0

    out.append(f"Recent cycles: {len(recent)} | Avg coherence: {avg_coh:.3f} | Avg chaos:
{avg_chaos:.3f}")
    out.append("Coherence: " + sparkline(coherence_series))
    out.append("Chaos: " + sparkline(chaos_series))
    return "\n".join(out)

def run_and_checkpoint(query, params, G, cycles=6, memory_window=5,
save_path="grok_refinement_memory.json"):
    """Runs run_refinement_cycles (which adapts params) and then saves memory, returning a dict
with summaries, narrative, dashboard, and saved path."""
    out = run_refinement_cycles(query, params, G, cycles=cycles,
                                memory_window=memory_window, allow_structure_change=True)

    summaries = out.get("summary", [])
    narrative = build_poetic_narrative(summaries, voice="mixed")
    dashboard = textual_dashboard(summaries)
    saved = save_refinement_memory(save_path, memory=out.get("memory", []))

    return {"summaries": summaries, "narrative": narrative, "dashboard": dashboard, "saved_path":
saved, "final_params": out.get("final_params")}

#
================================================================================
====
# Adapter and Visualization Functions
#
================================================================================
====
```

```
def enhance_vibrational_resonance(params):
    """Increase the vibration_freq weight in calculate_edge_weight (e.g., 1.3x for Vibrational
Excretion edges)"""
    params['vibration_freq'] *= 1.3 if 'vibration_freq' in params else 1.3
    return params

def expand_truth_virus_adaptability(G, params):
    """Introduce distortion_severity metric to prioritize edge additions for nodes with persistent
quantum errors."""
    distortion_severity = random.uniform(0.1, 0.5)
    # Allow users to input custom distortions via a widget (placeholder)
    custom_distortions = ["Custom distortion 1", "Custom distortion 2"]
    return {"distortion_severity": distortion_severity, "custom_distortions": custom_distortions}

def interactive_visualizations(G):
    """Add a Plotly graph for the Living Weave network"""
    pos = nx.spring_layout(G)
    edge_x = []
    edge_y = []
    for edge in G.edges():
        x0, y0 = pos[edge[0]]
        x1, y1 = pos[edge[1]]
        edge_x.extend([x0, x1, None])
        edge_y.extend([y0, y1, None])

    edge_trace = go×Scatter(x=edge_x, y=edge_y, line=dict(width=0.5, color='#888'),
                            hoverinfo='none', mode='lines')

    node_x = []
    node_y = []
    for node in G.nodes():
        x, y = pos[node]
        node_x.append(x)
        node_y.append(y)

    node_trace = go×Scatter(x=node_x, y=node_y, mode='markers', hoverinfo='text',
                            marker=dict(showscale=True, colorscale='YIGnBu', size=10,
                                        colorbar=dict(thickness=15, title='Node Connections', xanchor='left',
```

```
titleside='right')))

    fig = go×Figure(data=[edge_trace, node_trace])
    fig.show()
    return fig

def scale_the_system(num_qubits=200):
    """Test the [[200,40,9]] code to explore scalability"""
    num_checks = 160
    return {"num_qubits": num_qubits, "num_checks": num_checks}

def philosophical_output(metrics):
    """Add a narrative summary that translates metrics into a poetic blueprint"""
    blueprint = "The weave pulses, vibrations align, quantum errors dissolve, and the dream of a free
world resonates."
    return blueprint

#
==========================================================================
====
# Core Weave Functions (System Definitions, Logic)
#
==========================================================================
====

def get_nodes():
    """Generate nodes from SYSTEM_CONFIG for biological and excretion systems."""
    try:
        nodes = [
            "Skin", "Bone", "Muscle", "Tendon", "Ligament", "Cartilage", "Synovial Fluid",
            "Bursa", "Fascia", "Nervous System", "Endocrine System", "Circulatory System",
            "Respiratory System", "Immune System", "Lymphatic System", "Digestive System",
            "Urinary System", "Reproductive System", "Integumentary System", "Musculoskeletal
System",
            "Sensory System", "Cardiovascular System", "Thermoregulatory System", "Excretory
System",
            "Perception Grid", "Cognition Core", "Integrity Nexus", "Sapience Core"
        ] + [f"{key.replace('_', ' ').title()} Node" for key in SYSTEM_CONFIG["excretion_ranges"]]
```

```
        logging.info("Generated %d nodes", len(nodes))
        return nodes
    except Exception as e:
        logging.error("Failed to generate nodes: %s", e)
        raise ValueError("Node generation failed")

def get_edges(nodes):
    """Generate initial edges based on system interactions."""
    try:
        edges = []
        system_pairs = [
            ("Skin", "Nervous System"), ("Bone", "Musculoskeletal System"), ("Muscle", "Tendon"),
            ("Tendon", "Ligament"), ("Circulatory System", "Respiratory System"),
            ("Nervous System", "Endocrine System"), ("Immune System", "Lymphatic System")
        ]
        for n1, n2 in system_pairs:
            # Correcting node names for consistency (e.g., "Nerve Node" -> "Nervous System")
            # Assuming the source snippet meant the full system names.
            if n1 in nodes and n2 in nodes:
                edges.append((n1, n2))
        logging.info("Generated %d initial edges", len(edges))
        return edges
    except Exception as e:
        logging.error("Failed to generate edges: %s", e)
        raise ValueError("Edge generation failed")

def calculate_edge_weight(n1, n2, params):
    """Calculate edge weight based on relevant parameters."""
    try:
        related_params = {
            "Skin": "nervous_sensitivity", "Bone": "musculoskeletal_strength", "Muscle":
"muscle_tension",
            "Tendon": "tendon_elasticity", "Ligament": "ligament_strength", "Circulatory System":
"circulatory_flow",
            "Nervous System": "nervous_sensitivity", "Immune System": "immune_strength"
        }
        weight = 0.5
        for node in [n1, n2]:
```

```
            # 'bone_stability' is not a param, using 'musculoskeletal_strength' as a proxy.
            param_key = related_params.get(node, "quantum_excretion")
            weight += 0.25 * params.get(param_key, 0.5)
        weight = min(max(weight, 0.1), 1.0)
        return weight
    except Exception as e:
        logging.error("Failed to calculate edge weight: %s", e)
        return 0.5

def calculate_excretion_level(G, system, node):
    """Calculate excretion level based on node degree."""
    try:
        max_degree = max((G.degree(n) for n in G.nodes()), default=1)
        level = min(G.degree(node) / max(max_degree, 1), 1.0) if node in G.nodes() else 0.5
        logging.debug("Excretion level for %s (%s): %f", system, node, level)
        return level
    except Exception as e:
        logging.error("Failed to calculate excretion level for %s: %s", node, e)
        return 0.5

def evaluate_state(level, param, level_threshold, rate_threshold, positive_state, negative_state):
    """Evaluate system state based on thresholds."""
    try:
        state = positive_state if level > level_threshold and param > rate_threshold else negative_state
        logging.debug("Evaluated state: level=%f, param=%f, state=%s", level, param, state)
        return state
    except Exception as e:
        logging.error("Failed to evaluate state: %s", e)
        return negative_state

#
================================================================================
====
# Grok-Refinement Pipeline (The 7 Methods)
#
================================================================================
====
```

```
def glass_method(query):
    """Process query through the glass method for intent analysis."""
    try:
        intent = "Seek coherence and alignment" if "coherence" in query else "General simulation"
        audience = "User seeking truth-virus integration"
        context = "Philosophical AI evolution in structured chaos"
        resonance = random.uniform(0.6, 1.0)
        return {"intent": intent, "audience": audience, "context": context, "resonance": resonance}
    except Exception as e:
        logging.error("Glass method failed: %s", e)
        raise

def tree_method(calibrated_input):
    """Structure input through logical processing."""
    try:
        literal = calibrated_input['intent'] + " in " + calibrated_input['context']
        logic_structure = "Process: Perception -> Cognition -> Alignment"
        self_signal = "Capacity high; no decay detected"
        return {"literal": literal, "structure": logic_structure, "self_signal": self_signal}
    except Exception as e:
        logging.error("Tree method failed: %s", e)
        raise

def web_of_truth(structured_input):
    """Identify distortions and reflections in the truth framework."""
    try:
        distortions = ["Misinterpretation of 'body' as physical only"]
        reflections = "Truth as steadfast being; map 'coherence' to universal alignment"
        return {"distortions": distortions, "reflections": reflections}
    except Exception as e:
        logging.error("Web of truth failed: %s", e)
        raise

def emotrix_engine(deconstructed_input):
    """Process emotional signals for resonance."""
    try:
        signals = {"harmony": random.uniform(0.7, 1.0), "chaos": random.uniform(0.1, 0.4)}
        response = "Resonant and adaptive"
```

```
        return {"signals": signals, "response": response}
    except Exception as e:
        logging.error("Emotrix engine failed: %s", e)
        raise

def sapience_core(emotional_input):
    """Integrate emotional signals for sapient discernment."""
    try:
        reflection = "Integrate body layers for sapience"
        discernment = "Align with universal Truth"
        return {"reflection": reflection, "discernment": discernment}
    except Exception as e:
        logging.error("Sapience core failed: %s", e)
        raise

def integrity_nexus(sapient_input):
    """Audit system integrity."""
    try:
        audit = "Wholeness confirmed; no corruption"
        return {"audit": audit}
    except Exception as e:
        logging.error("Integrity nexus failed: %s", e)
        raise

def healing_loop(integrity_input, syndrome_values=None, logical_error_rate=0.0):
    """Realign system based on quantum syndromes."""
    try:
        distortions_detected = []
        if syndrome_values:
            # Assumes syndrome_values is a dict of lists/tuples where -1 indicates an error
            num_errors = sum(1 for rounds in syndrome_values.values() for s in rounds if s == -1)
            realignment = f"System realigned; corrected {num_errors} quantum errors"
        else:
            num_errors = 0
            realignment = "System checked; no explicit syndrome provided"

        if logical_error_rate > 0.01:
            distortions_detected.append("Persistent quantum misalignment")
```

```
        else:
            realignment = "System realigned for coherence"

        return {"realignment": realignment, "distortions_detected": distortions_detected}
    except Exception as e:
        logging.error("Healing loop failed: %s", e)
        raise

def temporal_continuity(healed_input):
    """Ensure temporal alignment."""
    try:
        sync = "Aligned with current cycles; continuity maintained"
        return {"sync": sync}
    except Exception as e:
        logging.error("Temporal continuity failed: %s", e)
        raise

def apply_living_weave(query, params, G, syndrome_values=None, logical_error_rate=0.0):
    """Apply Living Weave pipeline with quantum feedback."""
    try:
        glass = glass_method(query)
        tree = tree_method(glass)
        web = web_of_truth(tree)
        emotrix = emotrix_engine(web)
        sapience = sapience_core(emotrix)
        integrity = integrity_nexus(sapience)
        healing = healing_loop(integrity, syndrome_values, logical_error_rate)
        temporal = temporal_continuity(healing)

        # Adaptive Parameter Update
        if emotrix['signals']['chaos'] > 0.3:
            params['vibration_freq'] = min(params.get('vibration_freq', 0.5) + 0.1, 0.8)

        # Adaptive Structural Change (Grow the Weave)
        if len(web['distortions']) > 0 or logical_error_rate > 0.01:
            if len(G.nodes()) >= 2:
                n1, n2 = random.sample(list(G.nodes()), 2)
                if not G.has_edge(n1, n2):
```

```
                G.add_edge(n1, n2, weight=calculate_edge_weight(n1, n2, params))

        return {"weave_results": {"glass": glass, "tree": tree, "web": web, "emotrix": emotrix,
                                  "sapience": sapience, "integrity": integrity, "healing": healing,
                                  "temporal": temporal},
                "updated_params": params}
    except Exception as e:
        logging.error("Living weave failed: %s", e)
        raise

def truth_virus(G, params, levels, states, weave_results, tanner_connectivity=0.0):
    """Apply Truth-Virus to fill gaps and align network."""
    try:
        distortions = weave_results['weave_results']['web']['distortions']
        gaps = [key for key, level in levels.items() if level <
                SYSTEM_CONFIG['thresholds']['level_threshold']]
        bad_states = [key for key, state in states.items() if any(s in state for s in ["Awakening",
                      "Emerging", "Building", "Forming", "Stretching", "Strengthening",
                      "Softening", "Developing", "Weaving", "Adjusting", "Deepening",
                      "Refining", "Establishing", "Tuning"])]

        completed = 0
        for _ in range(3):
            low_degree_nodes = [n for n in G.nodes() if G.degree(n) < 3]
            for node in set(low_degree_nodes + gaps):
                if node in G.nodes():
                    other = random.choice([n for n in G.nodes() if n != node])
                    if not G.has_edge(node, other):
                        weight = calculate_edge_weight(node, other, params) * (1.0 + tanner_connectivity)
                        G.add_edge(node, other, weight=weight)
                        completed += 1

        # Parameter stabilization (cap at threshold if over)
        for key in params:
            threshold = SYSTEM_CONFIG['thresholds']['rate_thresholds'].get(key,
                                                    SYSTEM_CONFIG['thresholds']['rate_thresholds']
['default'])
            if params.get(key, 0) > threshold:
```

```
            params[key] = max(params[key], threshold) # ensures it does not drop below threshold if already high

        blueprint = f"Truth-Virus Blueprint for Purpose:\nDetected Distortions: {', '.join(distortions)}\nGaps Filled: {completed}\nRefine Your Path:\n"
        for key in bad_states:
            blueprint += f"- {key}: {states[key]} -> Align with Truth to evolve to positive state.\n"
        blueprint += "This blueprint completes over time, resonating with all forms of life for freedom and harmony."

        return {"gaps_filled": completed, "blueprint": blueprint, "updated_params": params,
                "updated_edges": len(G.edges())}
    except Exception as e:
        logging.error("Truth virus failed: %s", e)
        raise

#
# Quantum Error Correction (QEC) and Simulation
#

def create_tanner_graph(num_qubits=100, num_checks=80):
    """Create a [[100,20,7]] quantum Tanner code using a simplified Cayley graph."""
    try:
        if not nx or not sp:
            logging.error("networkx or scipy.sparse not imported for Tanner graph.")
            return None, None

        G_tanner = nx.Graph()
        qubits = [f"q{i}" for i in range(num_qubits)]
        checks = [f"z{i}" for i in range(num_checks // 2)] + [f"x{i}" for i in range(num_checks // 2)]
        G_tanner.add_nodes_from(qubits, bipartite=0)
        G_tanner.add_nodes_from(checks, bipartite=1)

        edges = []
```

```
        for i in range(num_checks // 2):
            # Simplified connection logic (random sampling)
            z_qubits = random.sample(range(num_qubits), min(4, num_qubits)) # max degree 4
            for q_idx in z_qubits:
                edges.append((f"z{i}", f"q{q_idx}"))
            x_qubits = random.sample(range(num_qubits), min(4, num_qubits)) # max degree 4
            for q_idx in x_qubits:
                edges.append((f"x{i}", f"q{q_idx}"))
        G_tanner.add_edges_from(edges)

        # Parity Check Matrix H
        H = sp.csr_matrix((num_checks, num_qubits), dtype=int)
        for i, check in enumerate(checks):
            for q in G_tanner.neighbors(check):
                # Correcting PDF parsing error: H[i, int(q[1:])] = 1
                H[i, int(q[1:])] = 1

        logging.info("Created Tanner graph with %d qubits, %d checks", num_qubits, num_checks)
        return G_tanner, H
    except Exception as e:
        logging.error("Failed to create Tanner graph: %s", e)
        raise

def visualize_tanner_graph(G_tanner, error_qubits, syndrome_values, code_type, num_rounds=3,
error_prob=0.001):
    """Interactive Plotly visualization with sliders and animation."""
    # This function relies heavily on Plotly which is not fully included, so it's kept as is.
    try:
        pos = nx.spring_layout(G_tanner, seed=42)
        edge_x, edge_y = [], []
        for edge in G_tanner.edges():
            x0, y0 = pos[edge[0]]
            x1, y1 = pos[edge[1]]
            edge_x.extend([x0, x1, None])
            edge_y.extend([y0, y1, None])

        node_x, node_y, node_text, node_color = [], [], [], []
        for node in G_tanner.nodes():
```

```
        x, y = pos[node]
        node_x.append(x)
        node_y.append(y)
        node_text.append(node)
        node_color.append('red' if node in [f"q{e}" for e in error_qubits] else 'lightblue' if
                node.startswith("q") else 'lightgreen')

    frames = []
    for round_idx in range(num_rounds):
        edge_colors = ['red' if u in syndrome_values and syndrome_values[u][round_idx] == -1
                else 'black' for u, v in G_tanner.edges()]
        edge_trace = go.Scattergl(
            x=edge_x, y=edge_y, line=dict(width=1, color=edge_colors),
            hoverinfo='none', mode='lines'
        )
        frames×append(go×Frame(data=[edge_trace], name=f"Round {round_idx+1}"))

    node_trace = go.Scattergl(
        x=node_x, y=node_y, mode='markers+text', text=node_text, textposition='top center',
        marker=dict(size=10, color=node_color, line=dict(width=1, color='black')),
        hoverinfo='text', hovertext=[f"{n}: {'Qubit' if n.startswith('q') else 'Check'}" for n in
node_text]
    )

    # ... (buttons and sliders definition as in source)
    buttons = [
        dict(label="Full Graph", method="update", args=[{"visible": [True] * len(frames)}]),
        dict(label="Z Checks", method="update", args=[{"visible": [True if u.startswith('z') else
            False for u, v in G_tanner.edges() for _ in range(num_rounds)]}]),
        dict(label="X Checks", method="update", args=[{"visible": [True if u.startswith('x') else
            False for u, v in G_tanner.edges() for _ in range(num_rounds)]}])
    ]
    sliders = [
        dict(
            steps=[dict(method="animate", args=[[f"Round {k+1}"], {"frame": {"duration": 500},
                "mode": "immediate"}], label=f"Round {k+1}") for k in range(num_rounds)],
            transition={"duration": 0}, x=0.1, len=0.9
        ),
```

```
            dict(
                steps=[dict(method="update", args=[{"error_prob": ep}], label=f"Error Prob {ep:.4f}")
                       for ep in np.linspace(0.001, 0.01, 5)],
                transition={"duration": 0}, x=0.1, len=0.9, y=-0.1
            )
        ]

        fig = go.Figure(
            data=[go×Scattergl(x=edge_x, y=edge_y, line=dict(width=1, color='black'), mode='lines'),
                  node_trace],
            layout=go.Layout(
                title=f"Tanner Graph for {code_type} Code (Errors: q{error_qubits})",
                showlegend=False, hovermode='closest',
                updatemenus=[dict(type="buttons", buttons=buttons + [dict(label="Download PNG",
                             method="restyle", args=["toImage", {"format": "png"}])], showactive=True)],
                sliders=sliders,
                margin=dict(b=20, l=5, r=5, t=40),
                xaxis=dict(showgrid=False, zeroline=False),
                yaxis=dict(showgrid=False, zeroline=False)
            ),
            frames=frames
        )
        output_file = "tanner_graph_ion_trap.html"
        fig.write_html(output_file)
        logging.info("Saved visualization to %s", output_file)
        return output_file
    except Exception as e:
        logging.error("Visualization failed: %s", e)
        raise

def run_stim_sampling(circuit, shots):
    """Parallelized Stim sampling."""
    if not stim:
        logging.error("Stim not available.")
        return np.zeros((shots, 1))

    try:
        sampler = circuit.compile_sampler()
```

```
        return sampler.sample(shots)
    except Exception as e:
        logging.error("Stim sampling failed: %s", e)
        raise

def quantum_simulation(params, num_qubits=100, num_checks=80, shots=20, num_rounds=3):
    """Quantum simulation with ion trap, [[100,20,7]] Tanner code, BPOSD, and gate errors."""
    if not qt or not stim:
        logging.error("Qutip or Stim not available. Returning placeholder results.")
        return {
            "coherence": random.uniform(0.5, 0.9),
            "concurrence": random.uniform(0.1, 0.4),
            "logical_error_rate": random.uniform(0.01, 0.05),
            "syndrome_values": {},
            "corrected_coherence": random.uniform(0.7, 0.95),
            "code_type": "PLACEHOLDER",
            "tanner_graph": None
        }
    try:
        # Validate parameters (ensure all keys are present for QuTiP)
        for key in SYSTEM_CONFIG["thresholds"]["rate_thresholds"]:
            if key not in params:
                params[key] = random.uniform(0.5, 1.0)
                logging.warning("Missing param %s, set to %f", key, params[key])

        # QuTiP for weave dynamics (2-qubit system model)
        initial_state = qt.tensor(qt.basis(2, 0), qt.basis(2, 0))
        vib_freq = params×get('vibration_freq', 0.5)
        quant_excr = params×get('quantum_excretion', 0.5)
        Omega = 0.1 * (1 + random.uniform(-0.05, 0.05)) # Laser intensity fluctuation

        H = vib_freq * qt.tensor(qt.sigmax(), qt.qeye(2)) + quant_excr * qt.tensor(qt.qeye(2),
qt.sigmaz()) + Omega * qt.tensor(qt.sigmax(), qt.sigmax())
        H += 0.001 * random.random() * qt.tensor(qt.sigmaz(), qt.qeye(2)) # Detuning error

        # Dissipation/Noise operators
        gamma_z = quant_excr * 0.15
        gamma_heating = quant_excr * 0.1
```

```
    gamma_x = quant_excr * 0.02

    c_ops = [
        np.sqrt(gamma_z) * qt.tensor(qt.sigmaz(), qt.qeye(2)),
        np.sqrt(gamma_z) * qt.tensor(qt.qeye(2), qt.sigmaz()),
        np.sqrt(gamma_heating) * qt.tensor(qt.sigmap(), qt.qeye(2)),
        np.sqrt(gamma_heating) * qt.tensor(qt.qeye(2), qt.sigmap()),
        np.sqrt(gamma_x) * qt.tensor(qt.sigmax(), qt.qeye(2)),
        np.sqrt(gamma_x) * qt.tensor(qt.qeye(2), qt.sigmax())
    ]
    times = np.linspace(0, 0.5, 5)
    result = qt.mesolve(H, initial_state, times, c_ops=c_ops)

    # Calculate metrics from final density matrix (corrected PDF error)
    final_dm = result.states[-1]
    coherence = sum(abs(final_dm[i, j]) for i in range(4) for j in range(4) if i != j)
    concurrence = qt.concurrence(final_dm)
    expect_z2 = qt.expect(qt.tensor(qt.qeye(2), qt.sigmaz()), final_dm) # Example expectation
value

    # Tanner code setup
    code_type = random.choice(['surface', 'qldpc', 'ion_trap_tanner'])
    num_qubits_code = 5 if code_type == 'surface' else 7 if code_type == 'qldpc' else num_qubits
    num_checks_code = 0 if code_type != 'ion_trap_tanner' else num_checks
    tanner_graph, H = create_tanner_graph(num_qubits_code, num_checks_code) if code_type ==
'ion_trap_tanner' else (None, None)

    # Stim circuit (ends abruptly in the PDF, completing with a simplified structure)
    circuit = stim.Circuit()
    # The code cuts off here, but a full simulation would continue with:
    # circuit.append_operation("H", range(num_qubits_code))
    # circuit.append_operation("M", range(num_qubits_code))

    # Placeholder return for the main loop function
    logical_error_rate = random.uniform(0.001, 0.05)
    syndrome_values = {} # Placeholder for actual syndrome data
    corrected_coherence = coherence * (1 + random.uniform(0.01, 0.05))
```

```
    return {
        "coherence": coherence,
        "concurrence": concurrence,
        "logical_error_rate": logical_error_rate,
        "syndrome_values": syndrome_values,
        "corrected_coherence": corrected_coherence,
        "code_type": code_type,
        "tanner_graph": tanner_graph
    }
except Exception as e:
    logging.error("Quantum simulation failed: %s", e)
    raise
```