```cpp
#include "brain_ai/evolve/metrics.hpp"
#include <sstream>
#include <iomanip>
#include <algorithm>

using namespace brain_ai::evolve;

Metrics& Metrics::instance() {
    static Metrics inst;
    return inst;
}

void Metrics::register_metric(
    const std::string& name,
    MetricType type,
    const std::string& help) {

    std::lock_guard<std::mutex> lock(mutex_);
    if (metrics_.find(name) != metrics_.end()) {
        return; // Already registered
    }

    MetricData data;
    data.type = type;
    data.help = help;
    data.value = 0.0;

    if (type == MetricType::HISTOGRAM) {
        // Default buckets: 0.001, 0.01, 0.1, 1, 10, 100
        data.histogram_buckets = {0.001, 0.01, 0.1, 1.0, 10.0, 100.0,
                                  std::numeric_limits<double>::infinity()};
        data.histogram_counts.resize(data.histogram_buckets.size(), 0);
    }
    metrics_[name] = data;
}

void Metrics::increment(const std::string& name, double value) {
    std::lock_guard<std::mutex> lock(mutex_);
    if (metrics_.find(name) == metrics_.end()) {
        register_metric(name, MetricType::COUNTER, "");
    }
    metrics_[name].value += value;
}

void Metrics::set_gauge(const std::string& name, double value) {
    std::lock_guard<std::mutex> lock(mutex_);
    if (metrics_.find(name) == metrics_.end()) {
        register_metric(name, MetricType::GAUGE, "");
    }
    metrics_[name].value = value;
}

void Metrics::observe_histogram(const std::string& name, double value) {
    std::lock_guard<std::mutex> lock(mutex_);
    if (metrics_.find(name) == metrics_.end()) {
        register_metric(name, MetricType::HISTOGRAM, "");
    }

    auto& data = metrics_[name];
    data.value += value; // Sum

    // Update histogram buckets
    for (size_t i = 0; i < data.histogram_buckets.size(); ++i) {
        if (value <= data.histogram_buckets[i]) {
            data.histogram_counts[i]++;
        }
```

```
    }
}

std::string Metrics::export_prometheus() const {
    std::lock_guard<std::mutex> lock(mutex_);
    std::ostringstream oss;

    for (const auto& [name, data] : metrics_) {
        // Help text
        if (!data.help.empty()) {
            oss << "# HELP " << name << " " << data.help << "\n";
        }

        // Type
        std::string type_str;
        switch (data.type) {
            case MetricType::COUNTER: type_str = "counter"; break;
            case MetricType::GAUGE: type_str = "gauge"; break;
            case MetricType::HISTOGRAM: type_str = "histogram"; break;
        }
        oss << "# TYPE " << name << " " << type_str << "\n";

        // Value
        if (data.type == MetricType::HISTOGRAM) {
            // Histogram format
            uint64_t cumulative_count = 0;
            for (size_t i = 0; i < data.histogram_buckets.size(); ++i) {
                cumulative_count += data.histogram_counts[i];
                oss << name << "_bucket{le=\"";
                if (std::isinf(data.histogram_buckets[i])) {
                    oss << "+Inf";
                } else {
                    oss << data.histogram_buckets[i];
                }
                oss << "\"} " << cumulative_count << "\n";
            }
            oss << name << "_sum " << data.value << "\n";
            oss << name << "_count " << cumulative_count << "\n";
        } else {
            oss << name << " " << std::fixed << std::setprecision(6) << data.value <<
"\n";
        }
    }
    return oss.str();
}
```

```
#include "brain_ai/api/brain_service.hpp"
#include "brain_ai/evolve/metrics.hpp"
#include <chrono>
#include <stdexcept>
#include <iostream>

#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
#include <sstream>

using namespace brain_ai::api;
using grpc::Status;
using grpc::ServerContext;

BrainServiceImpl::BrainServiceImpl(
    const std::string& audit_log_path,
    const std::string& keys_path) {

    // Initialize quantum workspace
    workspace::QuantumConfig config;
    config.rng_seed = 42;
    quantum_ = std::make_unique<workspace::QuantumStrict>(config);

    // Load signing key
    try {
        auto [pk, sk] = evolve::load_keypair(keys_path);
        public_key_ = pk;
        signing_key_ = sk;
    } catch (const std::exception& e) {
        std::cerr << "FATAL: Failed to load keypair from " << keys_path << ". " <<
e.what() << std::endl;
        throw;
    }

    // Initialize audit log
    audit_log_ = std::make_unique<evolve::MerkleAuditLog>(audit_log_path, keys_path +
"/public.key");
    if (!audit_log_->verify_chain(public_key_)) {
        std::cerr << "FATAL: Audit log verification failed on startup!" << std::endl;
        // In production, you might halt here.
    }

    // Register metrics
    auto& m = evolve::Metrics::instance();
    m.register_metric("brain_ai_step_total", evolve::MetricType::COUNTER, "Total quantum
steps executed");
    m.register_metric("brain_ai_entropy_nats", evolve::MetricType::GAUGE, "Current von
Neumann entropy");
    m.register_metric("brain_ai_trace_error", evolve::MetricType::GAUGE, "Density matrix
trace error");
    m.register_metric("brain_ai_step_duration_seconds", evolve::MetricType::HISTOGRAM,
"Step execution time");
}

Status BrainServiceImpl::Step(
    ServerContext* context,
    const StepRequest* request,
    StepResponse* response) {

    auto start = std::chrono::steady_clock::now();
    try {
        // Update global workspace if provided
        if (request->global_workspace_size() > 0) {
```

```
            std::vector<double> gw(request->global_workspace().begin(), request-
>global_workspace().end());
            // Dummy weights for projection
            std::vector<std::complex<double>> weights(60, {1.0 / std::sqrt(60.0), 0.0});
            quantum_->project_from_global_workspace(gw, weights);
        }

        // Execute step
        quantum_->step();

        // Perform measurement if requested
        if (request->measure()) {
            quantum_->measure_computational_basis();
            // Log to audit trail
            std::string payload = "{\"measured_state\":" + std::to_string(quantum_-
>measured_state()) + "}";
            audit_log_->append("measurement", payload, signing_key_);
        }

        // Populate response
        response->set_entropy(quantum_->von_neumann_entropy());
        response->set_measured_state(quantum_->measured_state());
        response->set_trace_error(quantum_->trace_error());
        response->set_is_valid(quantum_->is_positive_semidefinite() && quantum_-
>is_hermitian());

        // Update metrics
        auto& m = evolve::Metrics::instance();
        m.increment("brain_ai_step_total");
        m.set_gauge("brain_ai_entropy_nats", quantum_->von_neumann_entropy());
        m.set_gauge("brain_ai_trace_error", quantum_->trace_error());

        auto end = std::chrono::steady_clock::now();
        double duration = std::chrono::duration<double>(end - start).count();
        m.observe_histogram("brain_ai_step_duration_seconds", duration);

        return Status::OK;

    } catch (const std::exception& e) {
        return Status(grpc::StatusCode::INTERNAL, e.what());
    }
}

Status BrainServiceImpl::GetState(
    ServerContext* context,
    const GetStateRequest* request,
    GetStateResponse* response) {

    try {
        auto rho = quantum_->density_matrix();
        response->set_dimension(rho.rows());
        response->set_entropy(quantum_->von_neumann_entropy());

        // Flatten density matrix
        for (int i = 0; i < rho.rows(); ++i) {
            for (int j = 0; j < rho.cols(); ++j) {
                response->add_density_matrix_real(rho(i, j).real());
                response->add_density_matrix_imag(rho(i, j).imag());
            }
        }
        return Status::OK;
    } catch (const std::exception& e) {
        return Status(grpc::StatusCode::INTERNAL, e.what());
    }
}
```

```
Status BrainServiceImpl::StreamInference(
    ServerContext* context,
    grpc::ServerReader<InferenceInput>* reader,
    grpc::ServerWriter<InferenceOutput>* writer) {

    InferenceInput input;
    while (reader->Read(&input)) {
        auto start = std::chrono::steady_clock::now();
        InferenceOutput output;
        output.set_request_id(input.request_id());

        // Dummy inference (replace with real model)
        for (int i = 0; i < 10; ++i) {
            output.add_logits(i * 0.1);
        }

        auto end = std::chrono::steady_clock::now();
        auto latency = std::chrono::duration_cast<std::chrono::microseconds>(end -
start).count();
        output.set_latency_us(latency);
        writer->Write(output);
    }
    return Status::OK;
}

Status BrainServiceImpl::Health(
    ServerContext* context,
    const HealthRequest* request,
    HealthResponse* response) {

    response->set_status(HealthResponse::SERVING);
    response->set_version("2.0.0");

    // Calculate uptime
    static auto start_time = std::chrono::steady_clock::now();
    auto now = std::chrono::steady_clock::now();
    auto uptime = std::chrono::duration_cast<std::chrono::seconds>(now -
start_time).count();
    response->set_uptime_seconds(uptime);

    return Status::OK;
}

// --- MetricsServer Implementation ---

MetricsServer::MetricsServer(int port)
    : port_(port), server_fd_(-1), running_(false) {}

void MetricsServer::start() {
    server_fd_ = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd_ < 0) {
        throw std::runtime_error("Failed to create socket");
    }
    int opt = 1;
    setsockopt(server_fd_, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port_);

    if (bind(server_fd_, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
        throw std::runtime_error("Failed to bind socket");
    }
    if (listen(server_fd_, 10) < 0) {
        throw std::runtime_error("Failed to listen");
```

```
    }

    running_ = true;
    server_thread_ = std::thread(&MetricsServer::serve, this);
}

void MetricsServer::stop() {
    running_ = false;
    if (server_fd_ >= 0) {
        shutdown(server_fd_, SHUT_RDWR);
        close(server_fd_);
    }
    if (server_thread_.joinable()) {
        server_thread_.join();
    }
}

void MetricsServer::serve() {
    while (running_) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);
        int client_fd = accept(server_fd_, (struct sockaddr*)&client_addr, &client_len);
        if (client_fd < 0) continue;

        // Read request
        char buffer[4096];
        ssize_t n = read(client_fd, buffer, sizeof(buffer) - 1);
        if (n > 0) {
            buffer[n] = '\0';
            std::string request(buffer);
            // Handle request
            std::string response = handle_request(request);
            write(client_fd, response.data(), response.size());
        }
        close(client_fd);
    }
}

std::string MetricsServer::handle_request(const std::string& request) {
    // Check if GET /metrics
    if (request.find("GET /metrics") != std::string::npos) {
        auto metrics = evolve::Metrics::instance().export_prometheus();
        std::ostringstream response;
        response << "HTTP/1.1 200 OK\r\n";
        response << "Content-Type: text/plain; version=0.0.4\r\n";
        response << "Content-Length: " << metrics.size() << "\r\n";
        response << "\r\n";
        response << metrics;
        return response.str();
    }

    // 404
    std::string body = "Not Found";
    std::ostringstream response;
    response << "HTTP/1.1 404 Not Found\r\n";
    response << "Content-Length: " << body.size() << "\r\n";
    response << "\r\n";
    response << body;
    return response.str();
}
```

```
import grpc
import brain_ai_pb2 as pb
import brain_ai_pb2_grpc as svc
import numpy as np
from pathlib import Path

class BrainClient:
    """
    Client for the C++ Brain Kernel gRPC service.
    Handles secure mTLS communication.
    """
    def __init__(self, addr: str, tls_ca_cert: str, tls_client_cert: str, tls_client_key:
str):
        try:
            root = Path(tls_ca_cert).read_bytes()
            cert = Path(tls_client_cert).read_bytes()
            key = Path(tls_client_key).read_bytes()

            creds = grpc.ssl_channel_credentials(
                root_certificates=root,
                private_key=key,
                certificate_chain=cert
            )

            # Override target name for self-signed certs if needed
            options = (('grpc.ssl_target_name_override', 'brain.svc'),)

            chan = grpc.secure_channel(addr, creds, options=options)

            self.stub = svc.BrainAIStub(chan)
            print(f"BrainClient connected to {addr} with mTLS")
        except Exception as e:
            print(f"Failed to create secure channel: {e}")
            raise

    def step(self, gw=None, measure=False):
        req = pb.StepRequest()
        if gw is not None:
            req.global_workspace.extend([float(x) for x in gw])
        req.measure = bool(measure)
        return self.stub.Step(req)

    def get_state(self):
        return self.stub.GetState(pb.GetStateRequest())

    def health_check(self):
        try:
            req = pb.HealthRequest()
            resp = self.stub.Health(req, timeout=1.0)
            return resp.status == pb.HealthResponse.SERVING
        except Exception as e:
            print(f"Health check failed: {e}")
            return False
```

```
import pathlib
import subprocess
import tempfile
import os
import json
import shutil
from fnmatch import fnmatch

def _in_allowlist(path: str, allow: list[str], deny: list[str]) -> bool:
    """Checks if a path is permitted by the allow/deny globs."""
    if any(fnmatch(path, g) for g in deny):
        return False
    return any(fnmatch(path, g) for g in allow)

def apply_manifest(repo_root: str, manifest: dict, allow: list[str], deny: list[str]):
    """
    Applies the edits from a manifest to the filesystem within a chroot.
    This is the most dangerous part and MUST be heavily sandboxed.
    """
    root = pathlib.Path(repo_root).resolve()

    for e in manifest["edits"]:
        tgt_path = e["path"]
        tgt_full = (root / tgt_path).resolve()

        # Path traversal check
        if not str(tgt_full).startswith(str(root)):
            raise RuntimeError(f"Path escape detected: {tgt_path}")

        # Glob check
        if not _in_allowlist(tgt_path, allow, deny):
            raise RuntimeError(f"Blocked path: {tgt_path}")

        s = tgt_full.read_text(encoding="utf-8") if tgt_full.exists() else ""
        a, b = e["start"], e["end"]

        if e["op"] == "replace":
            s = s[:a] + e["text"] + s[b:]
        elif e["op"] == "insert":
            s = s[:a] + e["text"] + s[a:]
        elif e["op"] == "delete":
            s = s[:a] + s[b:]

        tgt_full.parent.mkdir(parents=True, exist_ok=True)
        tgt_full.write_text(s, encoding="utf-8")

def run_sandboxed(cmd: list[str], cwd: str, cpu_ms: int = 5000, mem_mb: int = 1024):
    """
    Runs tests in a minimal sandbox.
    In production, this should invoke the C++ SandboxRunner via gRPC
    or use cgroups/namespaces directly.
    """
    env = {k: v for k, v in os.environ.items() if k.startswith("PY") or k == "PATH"}

    try:
        r = subprocess.run(
            cmd,
            cwd=cwd,
            env=env,
            capture_output=True,
            text=True,
            timeout=cpu_ms / 1000.0,
            check=True # Throw on non-zero exit
        )
        return 0, r.stdout, r.stderr
    except subprocess.CalledProcessError as e:
```

```
    return e.returncode, e.stdout, e.stderr
except subprocess.TimeoutExpired as e:
    return -1, "", "TimeoutExpired"
```

```
import math
import random
import numpy as np

def _hash_to_vec(text: str, d: int) -> list[float]:
    """
    Deterministic stub to convert text to a 60D vector.
    Replace with a real sentence transformer for production.
    """
    h = 2166136261
    for c in text.encode():
        h = (h ^ c) * 16777619 & 0xFFFFFFFF

    rng = random.Random(h)
    v = [rng.random() - 0.5 for _ in range(d)]
    n = math.sqrt(sum(x * x for x in v)) or 1.0
    return [x / n for x in v]

def text_to_gw(s: str) -> list[float]:
    """Converts text to a 60D Global Workspace vector."""
    return _hash_to_vec(s, 60)

def embed_for_vdb(text: str, dim: int = 384) -> np.ndarray:
    """
    Fast deterministic hash embedding.
    Swap with Sentence Transformers later for higher recall.
    """
    h = 2166136261
    for c in text.encode():
        h = (h ^ c) * 16777619 & 0xFFFFFFFF

    rng = np.random.default_rng(h)
    v = rng.standard_normal(dim).astype(np.float32)
    n = np.linalg.norm(v) or 1.0
    return v / n
```

```
fastapi
uvicorn[standard]
toml
httpx
grpcio
protobuf
numpy
scipy
jsonschema
pynacl
# faiss-cpu # Install this manually if needed
pyflakes
pytest
```

```
import json
import asyncio
import numpy as np
from .llm_client import LLMClient
from .memory_vec import VectorMemory

SYS_PROMPT = """
Role: You are a safe AI planner. Your task is to plan a single, safe "brain step" based
on user input and retrieved memories.
Output: You MUST output a strict JSON object with keys {op, gw, measure, rationale}.
1.  `op`: Must be "brain.step" or "brain.measure". Default to "brain.step".
2.  `gw`: Must be a list of 60 floating-point numbers, representing a unit-norm vector.
3.  `measure`: Must be true or false. Default to false unless the user explicitly asks to
"measure" or "collapse".
4.  `rationale`: A brief explanation of your plan.
Use the retrieved context to refine the `gw` vector. Do not execute tools.
"""

def bounded_vec(v, n=60) -> list[float]:
    """Ensures the vector is size `n` and unit-norm."""
    if not isinstance(v, list): v = []

    # Pad or truncate
    v = (v + [0.0] * n)[:n]

    # Ensure all are floats
    v_float = [float(x) if isinstance(x, (int, float)) else 0.0 for x in v]

    # Normalize
    norm = np.linalg.norm(v_float) or 1.0
    return (np.array(v_float) / norm).tolist()

class CortexPlanner:
    """
    Uses an LLM to generate a plan, augmented by
    retrieval from vector memory.
    """
    def __init__(self, provider, model, mem_jsonl, mem_index, top_k=5):
        self.llm = LLMClient(provider, model)
        self.mem = VectorMemory(mem_jsonl, mem_index)
        self.top_k = top_k

    async def plan(self, user_text: str) -> dict:
        """Generates a plan from user text."""
        recalls = self.mem.search(user_text, self.top_k)

        context = {"query": user_text, "recall": recalls}
        prompt = (
            "User request and retrieved context:\n" +
            json.dumps(context, ensure_ascii=False, indent=2) +
            "\n\nProduce plan JSON now."
        )

        try:
            out = await self.llm.complete(SYS_PROMPT, prompt)
        except Exception as e:
            print(f"LLM planner failed: {e}")
            # Fallback to simple hash
            from .workspace_adapter import text_to_gw
            out = {
                "op": "brain.step",
                "gw": text_to_gw(user_text),
                "measure": False,
                "rationale": "LLM failed, using fallback hash."
            }
```

```
    # Validate and normalize the output
    plan = {
        "op": out.get("op", "brain.step"),
        "gw": bounded_vec(out.get("gw", [])),
        "measure": bool(out.get("measure", False)),
        "rationale": out.get("rationale", "")
    }

    # Ensure op is valid
    if plan["op"] not in ("brain.step", "brain.measure"):
        plan["op"] = "brain.step"

    return plan

def remember(self, text: str, meta: dict):
    """Adds an event to the vector memory."""
    self.mem.add(text, meta)
```

```
import uvicorn
import toml
import os
import asyncio
from fastapi import FastAPI, Header, HTTPException, Request
from pydantic import BaseModel
from contextlib import asynccontextmanager
import ssl

from brain_client import BrainClient
from policy_vm import PolicyVM
from critic import score
from memory import append as mem_append
from killswitch import enforce
from planner_llm import CortexPlanner
from self_editor.api import router as selfedit_router, init_self_editor

# --- Global State ---
CFG = {}
APP_STATE = {}

# --- Pydantic Models ---
class Thought(BaseModel):
    text: str
    measure: bool = False

# --- Lifespan Events (Startup/Shutdown) ---
@asynccontextmanager
async def lifespan(app: FastAPI):
    # --- Startup ---
    print("Starting Cortex...")
    global CFG, APP_STATE
    CFG = toml.load("config/cortex.toml")

    # Enforce kill switch at startup
    enforce(CFG["security"]["killswitch_file"])

    # Initialize Brain Kernel Client
    try:
        brain_client = BrainClient(
            addr=CFG["brain"]["address"],
            tls_ca_cert=CFG["brain"]["tls_ca_cert"],
            tls_client_cert=CFG["brain"]["tls_client_cert"],
            tls_client_key=CFG["brain"]["tls_client_key"]
        )
        if not brain_client.health_check():
            raise RuntimeError("Brain Kernel gRPC health check failed")
        APP_STATE["BRAIN"] = brain_client
    except Exception as e:
        print(f"FATAL: Could not connect to Brain Kernel. {e}")
        # In a real setup, this would crash-loop until the brain is up.
        os._exit(1) # Hard exit

    # Initialize Policy VM
    pol = PolicyVM(
        rules_path="config/policy_rules.json",
        require_signoff=CFG["security"]["require_policy_signoff"]
    )
    APP_STATE["POL"] = pol

    # Initialize LLM Planner
    planner = CortexPlanner(
        provider=CFG["llm"]["provider"],
        model=CFG["llm"]["model"],
        mem_jsonl=CFG["memory"]["path"],
        mem_index=CFG["memory"]["index"],
```

```
        top_k=CFG["memory"]["top_k"]
    )
    APP_STATE["PLN"] = planner

    # Initialize Self-Editor API
    self_edit_config = {
        "signing.kms_key_ref": CFG["signing"]["kms_key_ref"],
        "selfedit.root": CFG["selfedit"]["root"],
        "selfedit.allow_globs": CFG["selfedit"]["allow_globs"],
        "selfedit.deny_globs": CFG["selfedit"]["deny_globs"],
        "selfedit.max_bytes": CFG["selfedit"]["max_bytes"]
    }
    init_self_editor(self_edit_config, pol)

    print("Cortex started successfully.")
    yield
    # --- Shutdown ---
    print("Cortex shutting down...")

app = FastAPI(lifespan=lifespan)
app.include_router(selfedit_router)

# --- Middleware for Auth (Example) ---
@app.middleware("http")
async def check_auth_and_killswitch(request: Request, call_next):
    # Enforce kill switch on every request
    enforce(CFG["security"]["killswitch_file"])

    # mTLS is handled by the Uvicorn/Nginx layer.
    # We could add extra checks here, e.g., on client CN.
    # print(f"Client cert: {request.client.cert}")

    response = await call_next(request)
    return response

# --- API Endpoints ---
@app.post("/think")
async def think(thought: Thought):
    # 1. Get components from state
    brain: BrainClient = APP_STATE["BRAIN"]
    planner: CortexPlanner = APP_STATE["PLN"]
    pol: PolicyVM = APP_STATE["POL"]

    # 2. Plan with retrieval
    plan = await planner.plan(thought.text)
    if thought.measure:
        plan["measure"] = True

    # 3. Gates (Critic & Policy)
    ok, why = score(plan)
    if not ok: raise HTTPException(400, f"critic: {why}")

    ok, why = pol.verify(plan)
    if not ok: raise HTTPException(403, f"policy: {why}")

    # 4. Log and Remember
    mem_append({"event": "plan", "plan": plan, "text": thought.text})
    planner.remember(thought.text, {"kind": "user", "len": len(thought.text)})

    # 5. Execute brain step
    r = brain.step(plan["gw"], measure=plan["measure"])

    out = {
        "entropy": r.entropy,
        "measured_state": r.measured_state,
        "trace_error": r.trace_error,
```

```
        "valid": r.is_valid,
        "rationale": plan.get("rationale", "")
    }

    mem_append({"event": "result", "data": out})
    return out

@app.get("/state")
async def state():
    brain: BrainClient = APP_STATE["BRAIN"]
    s = brain.get_state()
    return {"dimension": s.dimension, "entropy": s.entropy}

@app.post("/kill")
async def kill():
    arm(CFG["security"]["killswitch_file"])
    return {"status": "armed"}

@app.post("/unkill")
async def unkill():
    disarm(CFG["security"]["killswitch_file"])
    return {"status": "disarmed"}

# --- Main execution ---
def main():
    cfg = toml.load("config/cortex.toml")

    # Setup SSL context for mTLS
    ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    ssl_context.load_cert_chain(
        certfile=cfg["tls"]["server_cert"],
        keyfile=cfg["tls"]["server_key"]
    )
    ssl_context.load_verify_locations(cafile=cfg["tls"]["ca"])
    ssl_context.verify_mode = ssl.CERT_REQUIRED

    uvicorn.run(
        "main:app",
        host=cfg["server"]["host"],
        port=cfg["server"]["port"],
        ssl_keyfile=cfg["tls"]["server_key"],
        ssl_certfile=cfg["tls"]["server_cert"],
        ssl_ca_certs=cfg["tls"]["ca"],
        ssl_cert_reqs=ssl.CERT_REQUIRED
    )

if __name__ == "__main__":
    # Create dummy certs/keys for brain-client if they don't exist
    # (In prod, these are mounted)
    p_ca = pathlib.Path(CFG.get("brain", {}).get("tls_ca_cert", "ca.crt"))
    p_cert = pathlib.Path(CFG.get("brain", {}).get("tls_client_cert", "client.crt"))
    p_key = pathlib.Path(CFG.get("brain", {}).get("tls_client_key", "client.key"))

    if not p_ca.exists():
        print("Warning: Dummy brain certs not found. Creating.")
        p_ca.parent.mkdir(parents=True, exist_ok=True)
        p_ca.write_text("dummy_ca")
        p_cert.write_text("dummy_cert")
        p_key.write_text("dummy_key")

    main()
```

```cpp
#include "brain_ai/evolve/merkle_log.hpp"
#include <gtest/gtest.h>
#include <fstream>
#include <cstdio> // for std::remove
#include <sys/stat.h> // for rmdir

using namespace brain_ai::evolve;

class MerkleLogTest : public ::testing::Test {
protected:
    void SetUp() override {
        test_log_ = "/tmp/test_audit.log";
        test_keys_ = "/tmp/test_keys";

        // Clean up any previous test files
        std::remove(test_log_.c_str());
        std::remove((test_keys_ + "/public.key").c_str());
        std::remove((test_keys_ + "/secret.key").c_str());
        rmdir(test_keys_.c_str());

        // Generate test keypair
        auto [pk, sk] = ed25519_keypair();
        save_keypair(test_keys_, pk, sk);
        public_key_ = pk;
        secret_key_ = sk;
    }

    void TearDown() override {
        std::remove(test_log_.c_str());
        std::remove((test_keys_ + "/public.key").c_str());
        std::remove((test_keys_ + "/secret.key").c_str());
        rmdir(test_keys_.c_str());
    }

    std::string test_log_;
    std::string test_keys_;
    std::vector<uint8_t> public_key_;
    std::vector<uint8_t> secret_key_;
};

TEST_F(MerkleLogTest, KeypairGeneration) {
    EXPECT_EQ(public_key_.size(), 32u); // crypto_sign_PUBLICKEYBYTES
    EXPECT_EQ(secret_key_.size(), 64u); // crypto_sign_SECRETKEYBYTES

    bool pk_nonzero = false;
    for (uint8_t b : public_key_) {
        if (b != 0) pk_nonzero = true;
    }
    EXPECT_TRUE(pk_nonzero);
}

TEST_F(MerkleLogTest, SignatureVerification) {
    std::string message = "test message";
    std::vector<uint8_t> msg_bytes(message.begin(), message.end());
    auto signature = ed25519_sign(msg_bytes, secret_key_);
    EXPECT_TRUE(ed25519_verify(msg_bytes, signature, public_key_));

    // Wrong message should fail
    std::string wrong = "wrong message";
    std::vector<uint8_t> wrong_bytes(wrong.begin(), wrong.end());
    EXPECT_FALSE(ed25519_verify(wrong_bytes, signature, public_key_));

    // Tampered signature should fail
    signature[0] ^= 0xFF;
    EXPECT_FALSE(ed25519_verify(msg_bytes, signature, public_key_));
}
```

```
TEST_F(MerkleLogTest, AppendAndVerify) {
    MerkleAuditLog log(test_log_, test_keys_);
    auto entry1 = log.append("test_event", "{\"data\":1}", secret_key_);
    EXPECT_EQ(entry1.index, 0u);
    EXPECT_EQ(entry1.event_type, "test_event");

    auto entry2 = log.append("another_event", "{\"data\":2}", secret_key_);
    EXPECT_EQ(entry2.index, 1u);

    // Verify chain
    EXPECT_TRUE(log.verify_chain(public_key_));
}

TEST_F(MerkleLogTest, ChainIntegrity) {
    MerkleAuditLog log(test_log_, test_keys_);
    for (int i = 0; i < 10; ++i) {
        std::string payload = "{\"step\":" + std::to_string(i) + "}";
        log.append("step", payload, secret_key_);
    }
    EXPECT_TRUE(log.verify_chain(public_key_));

    // Tamper with file
    {
        std::fstream file(test_log_, std::ios::in | std::ios::out | std::ios::binary);
        file.seekp(100);
        char tamper = 0xFF;
        file.write(&tamper, 1);
    }
    // Verification should fail
    EXPECT_FALSE(log.verify_chain(public_key_));
}

TEST_F(MerkleLogTest, PersistenceAndReload) {
    {
        MerkleAuditLog log(test_log_, test_keys_);
        log.append("event1", "{}", secret_key_);
        log.append("event2", "{}", secret_key_);
    }
    // Reload
    MerkleAuditLog log2(test_log_, test_keys_);
    auto entries = log2.read_all();
    EXPECT_EQ(entries.size(), 2u);
    EXPECT_EQ(entries[0].event_type, "event1");
    EXPECT_EQ(entries[1].event_type, "event2");
    EXPECT_TRUE(log2.verify_chain(public_key_));
}

TEST_F(MerkleLogTest, HashChaining) {
    MerkleAuditLog log(test_log_, test_keys_);
    auto e1 = log.append("e1", "{}", secret_key_);
    auto e2 = log.append("e2", "{}", secret_key_);
    auto e3 = log.append("e3", "{}", secret_key_);

    // Each entry's previous hash should match previous entry's current hash
    EXPECT_EQ(e2.previous_hash, e1.current_hash);
    EXPECT_EQ(e3.previous_hash, e2.current_hash);
}
```

FDQC Brain-AI Project

This repository contains the complete, production-grade implementation of the Finite-Dimensional Quantum Consciousness (FDQC) framework. It is divided into two primary components:

1. Brain Kernel (/brain-kernel): A hardened, high-performance C++ application that implements the core n=7 quantum workspace (QuantumStrict), a cryptographically-signed MerkleAuditLog (via libsodium), a SeccompSandbox for isolated execution, and a secure gRPC service for control. This is the "brainstem."
2. Cortex (/cortex): A high-level Python application that acts as the "prefrontal cortex." It controls the Brain Kernel via a secure mTLS gRPC client. It includes:
   - A FastAPI server for mTLS-secured external commands.
   - An LLMClient to plan actions (e.g., OpenAI, Anthropic).
   - A VectorMemory (using FAISS) for retrieval-augmented planning.
   - A PolicyVM to enforce safety rules on all actions.
   - A SelfEditor API (/selfedit/*) that allows the AI to safely propose, sign, and apply edits to its own codebase, gated by CI tests and cryptographic approvals.

Theoretical Basis

This code is the implementation of the theories described in the provided documents:

- Document.txt: Describes the n=7 Finite-Dimensional Quantum Consciousness, the 60D Global Workspace, and the C++ "Brain Kernel" architecture.
- OW4X1-fractal_dynamic_quantum_consciousness_fdqc_v4.md: Describes the 8-subsystem architecture, thermodynamic principles ($E \propto n^2$), and variable capacity. This code implements the n=7 core described in the text file, with the variable capacity and other subsystems managed by the Python Cortex.

Quick Start

WARNING: This is a complex, production-grade system. It requires Docker, C++ build tools, Python, and a deep understanding of security.

1. Generate Security Artifacts

Keys & Certs: You MUST generate the mTLS certificates and signing keys first.

```
# 1. Generate mTLS certs (ca.crt, brain.key, cortex.crt, etc.)
bash scripts/generate_certs.sh

# 2. Generate Ed25519 keys for the audit log
# (You'll need a simple C++ or Python script using libsodium/pynacl)
# For now, we assume they are pre-provisioned.
mkdir -p brain-kernel/keys
mkdir -p brain-kernel/certs
cp certs/ca.crt certs/brain.crt certs/brain.key brain-kernel/certs/
# Copy keys to /etc/brain_ai/keys in the container or local dev
```

2. Build and Run the C++ Brain Kernel

You can build and run the brain_ai_server binary directly or use the provided Dockerfile.

```
# Navigate to the C++ kernel
cd brain-kernel

# Configure and Build
cmake -B build -S . -GNinja -DCMAKE_BUILD_TYPE=Release
cmake --build build

# Run the server (ensure certs/keys are in the right place)
# This requires sudo to access /etc/ dirs, or change paths in main.cpp
./build/brain_ai_server --keys_dir=./keys --certs_dir=./certs
```

3. Run the Python Cortex

The Cortex connects to the Brain Kernel.

```
# Navigate to the Python cortex
cd cortex

# Create a virtual environment and install dependencies
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

# Set API keys for the LLM
export OPENAI_API_KEY="sk-..."

# Run the Cortex server
# It will load its certs from config/cortex.toml
python3 -m src.main
```

4. Interact with the AI

You must use the operator.crt and operator.key client certificates to communicate with the Cortex's mTLS-protected API.

```
# Example: Send a "thought" to the AI
curl -s -X POST https://localhost:8080/think \
    --cert certs/operator.crt \
    --key certs/operator.key \
    --cacert certs/ca.crt \
    -H "Content-Type: application/json" \
    -d '{
        "text": "First thought: stabilize and report status.",
        "measure": false
    }' | jq
```

5. Check Observability

- Prometheus Metrics: curl http://localhost:9090/metrics
- Audit Log: brain_ai_audit.log will be populated with cryptographically-signed entries.
- Episodic Memory: cortex/memory_episodic.jsonl will log all thoughts and plans.

```
def score(plan: dict) -> tuple[bool, str]:
    """
    A simple "critic" that validates the basic structure
    of a plan before it's sent to the PolicyVM.
    """
    op = plan.get("op")
    if op not in ("brain.step", "brain.measure"):
        return False, "invalid-op"

    gw = plan.get("gw", [])
    if not isinstance(gw, list) or len(gw) > 60:
        return False, "gw-size"

    if any((not isinstance(x, (int, float)) or abs(x) > 10) for x in gw):
        return False, "gw-range"

    return True, ""
```

```
#pragma once

#include <string>
#include <vector>
#include <functional>
#include <sys/types.h>

namespace brain_ai {
namespace evolve {

// Configuration for the seccomp-bpf sandbox
struct SandboxConfig {
    uint64_t timeout_ms = 5000;
    uint64_t max_memory_bytes = 256 * 1024 * 1024; // 256 MB
    uint64_t max_cpu_time_ms = 3000;
    bool enable_network = false;
    std::string chroot_dir = "/tmp/sandbox";
    std::vector<std::string> allowed_syscalls = {
        "read", "write", "exit", "exit_group", "brk", "mmap", "munmap"
    };
};

// Result of a sandboxed execution
struct SandboxResult {
    int exit_code;
    std::string stdout_output;
    std::string stderr_output;
    uint64_t elapsed_ms;
    uint64_t memory_peak_bytes;
    bool killed_timeout;
    bool killed_violation;
    std::string violation_reason;
};

// Runs an executable in a hardened, isolated environment using
// Linux namespaces, cgroups, and seccomp-bpf.
class SandboxRunner {
public:
    explicit SandboxRunner(const SandboxConfig& config = {});

    // Execute code in isolated environment
    SandboxResult execute(
        const std::string& executable_path,
        const std::vector<std::string>& args = {}
    );

    // Execute with stdin
    SandboxResult execute_with_stdin(
        const std::string& executable_path,
        const std::string& stdin_data,
        const std::vector<std::string>& args = {}
    );

private:
    SandboxConfig config_;

    void setup_seccomp_filter();
    void setup_namespaces();
    void setup_cgroups(pid_t pid);
    void setup_chroot();
    void drop_privileges();
};

// Policy verifier: checks if a plan is allowed
class PolicyVerifier {
public:
```

```
    bool verify_plan(const std::string& plan_json);
    void add_rule(const std::string& rule_pattern);

private:
    std::vector<std::string> allowlist_patterns_;
};

} // namespace evolve
} // namespace brain_ai
```

```
#include "brain_ai/workspace/quantum_strict.hpp"
#include <iostream>
#include <cmath>

using namespace brain_ai::workspace;
using Eigen::MatrixXcd;
using Eigen::VectorXcd;
using Eigen::VectorXd;

QuantumStrict::QuantumStrict(const QuantumConfig& cfg)
    : config_(cfg),
      rho_(cfg.dimension, cfg.dimension),
      H_(cfg.dimension, cfg.dimension),
      entropy_(0.0),
      measured_state_(-1),
      current_dt_(cfg.dt),
      rng_(cfg.rng_seed) {

    // Initialize to maximally mixed state: rho = I / d
    rho_ = MatrixXcd::Identity(config_.dimension, config_.dimension) /
static_cast<double>(config_.dimension);
    entropy_ = std::log(static_cast<double>(config_.dimension));

    // Generate a random Hermitian Hamiltonian
    std::normal_distribution<double> dist(0.0, 0.1);
    H_.setZero();
    for (int i = 0; i < config_.dimension; ++i) {
        for (int j = i; j < config_.dimension; ++j) {
            double re = dist(rng_);
            double im = (i == j) ? 0.0 : dist(rng_);
            std::complex<double> val(re, im);
            H_(i, j) = val;
            H_(j, i) = std::conj(val);
        }
    }

    // Initialize Lindblad operators: dephasing in computational basis
    L_.clear();
    double gamma = config_.decoherence_rate;
    for (int k = 0; k < config_.dimension; ++k) {
        MatrixXcd Lk = MatrixXcd::Zero(config_.dimension, config_.dimension);
        Lk(k, k) = std::sqrt(gamma);
        L_.push_back(Lk);
    }

    // Pre-compute Kraus operators for the first step
    update_kraus_operators();
}

void QuantumStrict::update_kraus_operators() {
    // Pre-compute Kraus operators for CPTP map
    // E0 = sqrt(I - dt * sum(Lk_dag * Lk)) * U
    // Ek = sqrt(dt) * Lk * U
    // This is a first-order Trotter expansion.
    K_.clear();

    // Unitary part
    MatrixXcd U = (-std::complex<double>(0, 1) * current_dt_ * H_).exp();

    // Dissipative correction factor
    MatrixXcd correction = MatrixXcd::Identity(config_.dimension, config_.dimension);
    for (const auto& Lk : L_) {
        correction -= 0.5 * current_dt_ * (Lk.adjoint() * Lk);
    }

    // Get sqrt(correction) via eigendecomposition
```

```
    Eigen::SelfAdjointEigenSolver<MatrixXcd> es(correction);
    MatrixXcd sqrt_correction = es.eigenvectors() *
es.eigenvalues().cwiseSqrt().asDiagonal() * es.eigenvectors().adjoint();

    K_.push_back(sqrt_correction * U); // K_0

    // Dissipative Kraus operators
    for (const auto& Lk : L_) {
        K_.push_back(Lk * std::sqrt(current_dt_) * U); // K_k
    }
}


void QuantumStrict::evolve_cptp_kraus() {
    // Apply Kraus map: rho' = sum(Ek * rho * Ek_dag)
    // This is CPTP by construction if sum(Ek_dag * Ek) = I
    MatrixXcd rho_new = MatrixXcd::Zero(config_.dimension, config_.dimension);
    for (const auto& Ek : K_) {
        rho_new += Ek * rho_ * Ek.adjoint();
    }
    rho_ = rho_new;
}

void QuantumStrict::enforce_cptp_projection() {
    // Project to physical subspace if numerics drift
    // 1. Hermitianize
    rho_ = 0.5 * (rho_ + rho_.adjoint());

    // 2. Eigendecomposition on COMPLEX Hermitian matrix
    Eigen::SelfAdjointEigenSolver<MatrixXcd> es(rho_);
    if (es.info() != Eigen::Success) {
        throw std::runtime_error("Eigendecomposition failed");
    }

    // 3. Clip negative eigenvalues
    VectorXd eval = es.eigenvalues().real();
    for (int i = 0; i < eval.size(); ++i) {
        eval(i) = std::max(eval(i), config_.eigenvalue_floor);
    }

    // 4. Renormalize to trace 1
    double trace = eval.sum();
    if (trace > 0) {
        eval /= trace;
    }

    // 5. Reconstruct with complex eigenvectors
    rho_ = es.eigenvectors() * eval.asDiagonal() * es.eigenvectors().adjoint();
}

double QuantumStrict::compute_entropy_exact() const {
    // Von Neumann entropy: S(rho) = -Tr(rho * log(rho)) = -sum(lambda_i * log(lambda_i))
    // Use complex Hermitian eigensolver
    Eigen::SelfAdjointEigenSolver<MatrixXcd> es(rho_);
    if (es.info() != Eigen::Success) {
        return -1.0; // Signal error
    }

    double s = 0.0;
    const auto& eigenvalues = es.eigenvalues();
    for (int i = 0; i < eigenvalues.size(); ++i) {
        double lambda = eigenvalues(i).real();
        if (lambda > config_.eigenvalue_floor) {
            s -= lambda * std::log(lambda);
        }
    }
```

```
    return s;
}

bool QuantumStrict::is_positive_semidefinite() const {
    return min_eigenvalue() >= -config_.eigenvalue_floor;
}

double QuantumStrict::min_eigenvalue() const {
    Eigen::SelfAdjointEigenSolver<MatrixXcd> es(rho_);
    if (es.info() != Eigen::Success) return -999.0;
    return es.eigenvalues().real().minCoeff();
}

void QuantumStrict::adaptive_timestep() {
    if (!config_.adaptive_step) return;

    // Simple adaptive scheme: reduce dt if trace error grows
    double trace_err = trace_error();
    if (trace_err > 10 * config_.trace_tolerance) {
        current_dt_ = std::max(current_dt_ * 0.5, config_.dt_min);
        update_kraus_operators(); // Re-compute operators for new dt
    } else if (trace_err < config_.trace_tolerance) {
        current_dt_ = std::min(current_dt_ * 1.1, config_.dt_max);
        update_kraus_operators(); // Re-compute operators for new dt
    }
}

void QuantumStrict::step() {
    evolve_cptp_kraus();

    // Check if projection needed
    if (trace_error() > config_.trace_tolerance || !is_positive_semidefinite()) {
        enforce_cptp_projection();
    }

    entropy_ = compute_entropy_exact();
    adaptive_timestep();
    measured_state_ = -1; // Reset measurement
}

void QuantumStrict::measure_computational_basis() {
    // Explicit measurement via POVM {|k><k|}
    std::vector<double> probabilities(config_.dimension);
    double sum = 0.0;
    for (int k = 0; k < config_.dimension; ++k) {
        probabilities[k] = std::max(0.0, rho_(k, k).real());
        sum += probabilities[k];
    }

    // Normalize
    if (sum > 0) {
        for (auto& p : probabilities) p /= sum;
    } else {
        std::fill(probabilities.begin(), probabilities.end(), 1.0 / config_.dimension);
    }

    // Sample outcome
    std::discrete_distribution<int> dist(probabilities.begin(), probabilities.end());
    measured_state_ = dist(rng_);

    // Apply projection: rho -> |k><k|
    rho_ = MatrixXcd::Zero(config_.dimension, config_.dimension);
    rho_(measured_state_, measured_state_) = 1.0;
    entropy_ = 0.0;
}
```

```
void QuantumStrict::project_from_global_workspace(
    const std::vector<double>& gw,
    const std::vector<std::complex<double>>& learned_weights) {

    // Learned linear map: |psi> = sum_i(w_i * gw_component_i)
    if (learned_weights.size() != gw.size()) {
        throw std::invalid_argument("Weight dimension mismatch");
    }

    VectorXcd psi = VectorXcd::Zero(config_.dimension);
    for (size_t i = 0; i < gw.size() && i < learned_weights.size(); ++i) {
        int target_dim = i % config_.dimension;
        psi(target_dim) += learned_weights[i] * gw[i];
    }

    // Normalize
    double norm = psi.norm();
    if (norm > 1e-10) {
        psi /= norm;
    } else {
        // Fallback to uniform
        psi.setConstant(1.0 / std::sqrt(static_cast<double>(config_.dimension)));
    }

    // Construct density matrix rho = |psi><psi|
    rho_ = psi * psi.adjoint();
    entropy_ = compute_entropy_exact();
    measured_state_ = -1;
}
```

```
# --- Builder Stage ---
FROM ubuntu:22.04 AS builder

RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    ninja-build \
    git \
    pkg-config \
    libeigen3-dev \
    libsodium-dev \
    libseccomp-dev \
    libgrpc++-dev \
    libprotobuf-dev \
    protobuf-compiler-grpc \
    ca-certificates \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /build

COPY . .

RUN cmake -B build -S . -GNinja \
    -DCMAKE_BUILD_TYPE=Release \
    -DENABLE_TESTS=OFF \
    -DENABLE_BENCHMARKS=OFF \
    -DCMAKE_INSTALL_PREFIX=/usr/local

RUN cmake --build build --target install

# --- Production Image ---
# Use distroless for a minimal attack surface
FROM gcr.io/distroless/cc-debian12

COPY --from=builder /usr/local/bin/brain_ai_server /usr/local/bin/
COPY --from=builder /usr/local/lib/libbrain_ai.so.2 /usr/local/lib/
COPY --from=builder /lib/x86_64-linux-gnu/libsodium.so.23 /lib/x86_64-linux-gnu/
COPY --from=builder /lib/x86_64-linux-gnu/libseccomp.so.2 /lib/x86_64-linux-gnu/
# Add other shared libraries as needed (use ldd)

# Copy keys and certs
# In a real setup, these would be mounted via Kubernetes secrets
COPY --from=builder /etc/brain_ai/keys /etc/brain_ai/keys
COPY --from=builder /etc/brain_ai/certs /etc/brain_ai/certs

# Create non-root user
USER 65534:65534

# Expose ports
EXPOSE 50051 9090

# Run server
ENTRYPOINT ["/usr/local/bin/brain_ai_server"]
CMD ["--grpc_port=50051", "--metrics_port=9090"]
```

```
# Generated by the gRPC Python protocol compiler plugin. DO NOT EDIT!
"""Client and server classes corresponding to protobuf-defined services."""
import grpc

import brain_ai_pb2 as brain__ai__pb2


class BrainAIStub(object):
    """The main gRPC service for the Brain Kernel
    """

    def __init__(self, channel):
        """Constructor.

        Args:
            channel: A grpc.Channel.
        """
        self.Step = channel.unary_unary(
                '/brain_ai.BrainAI/Step',
                request_serializer=brain__ai__pb2.StepRequest.SerializeToString,
                response_deserializer=brain__ai__pb2.StepResponse.FromString,
                )
        self.GetState = channel.unary_unary(
                '/brain_ai.BrainAI/GetState',
                request_serializer=brain__ai__pb2.GetStateRequest.SerializeToString,
                response_deserializer=brain__ai__pb2.GetStateResponse.FromString,
                )
        self.StreamInference = channel.stream_stream(
                '/brain_ai.BrainAI/StreamInference',
                request_serializer=brain__ai__pb2.InferenceInput.SerializeToString,
                response_deserializer=brain__ai__pb2.InferenceOutput.FromString,
                )
        self.Health = channel.unary_unary(
                '/brain_ai.BrainAI/Health',
                request_serializer=brain__ai__pb2.HealthRequest.SerializeToString,
                response_deserializer=brain__ai__pb2.HealthResponse.FromString,
                )


class BrainAIServicer(object):
    """The main gRPC service for the Brain Kernel
    """

    def Step(self, request, context):
        """Executes one step of the quantum evolution
        """
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')

    def GetState(self, request, context):
        """Gets the current state of the density matrix
        """
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')

    def StreamInference(self, request_iterator, context):
        """Bidirectional streaming for inference (example)
        """
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')

    def Health(self, request, context):
        """Health check
```

```
        """
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')


def add_BrainAIServicer_to_server(servicer, server):
    rpc_method_handlers = {
            'Step': grpc.unary_unary_rpc_method_handler(
                    servicer.Step,
                    request_deserializer=brain__ai__pb2.StepRequest.FromString,
                    response_serializer=brain__ai__pb2.StepResponse.SerializeToString,
            ),
            'GetState': grpc.unary_unary_rpc_method_handler(
                    servicer.GetState,
                    request_deserializer=brain__ai__pb2.GetStateRequest.FromString,

response_serializer=brain__ai__pb2.GetStateResponse.SerializeToString,
            ),
            'StreamInference': grpc.stream_stream_rpc_method_handler(
                    servicer.StreamInference,
                    request_deserializer=brain__ai__pb2.InferenceInput.FromString,
                    response_serializer=brain__ai__pb2.InferenceOutput.SerializeToString,
            ),
            'Health': grpc.unary_unary_rpc_method_handler(
                    servicer.Health,
                    request_deserializer=brain__ai__pb2.HealthRequest.FromString,
                    response_serializer=brain__ai__pb2.HealthResponse.SerializeToString,
            ),
    }
    generic_handler = grpc.method_handlers_generic_handler(
            'brain_ai.BrainAI', rpc_method_handlers)
    server.add_generic_rpc_handlers((generic_handler,))


 # This class is part of an EXPERIMENTAL API.
class BrainAI(object):
    """The main gRPC service for the Brain Kernel
    """

    @staticmethod
    def Step(request,
            target,
            options=(),
            channel_credentials=None,
            call_credentials=None,
            insecure=False,
            compression=None,
            wait_for_ready=None,
            timeout=None,
            metadata=None):
        return grpc.experimental.unary_unary(request, target, '/brain_ai.BrainAI/Step',
            brain__ai__pb2.StepRequest.SerializeToString,
            brain__ai__pb2.StepResponse.FromString,
            options, channel_credentials,
            insecure, call_credentials, compression, wait_for_ready, timeout, metadata)

    @staticmethod
    def GetState(request,
            target,
            options=(),
            channel_credentials=None,
            call_credentials=None,
            insecure=False,
            compression=None,
            wait_for_ready=None,
```

```
            timeout=None,
            metadata=None):
        return grpc.experimental.unary_unary(request, target,
'/brain_ai.BrainAI/GetState',
            brain__ai__pb2.GetStateRequest.SerializeToString,
            brain__ai__pb2.GetStateResponse.FromString,
            options, channel_credentials,
            insecure, call_credentials, compression, wait_for_ready, timeout, metadata)

    @staticmethod
    def StreamInference(request_iterator,
            target,
            options=(),
            channel_credentials=None,
            call_credentials=None,
            insecure=False,
            compression=None,
            wait_for_ready=None,
            timeout=None,
            metadata=None):
        return grpc.experimental.stream_stream(request_iterator, target,
'/brain_ai.BrainAI/StreamInference',
            brain__ai__pb2.InferenceInput.SerializeToString,
            brain__ai__pb2.InferenceOutput.FromString,
            options, channel_credentials,
            insecure, call_credentials, compression, wait_for_ready, timeout, metadata)

    @staticmethod
    def Health(request,
            target,
            options=(),
            channel_credentials=None,
            call_credentials=None,
            insecure=False,
            compression=None,
            wait_for_ready=None,
            timeout=None,
            metadata=None):
        return grpc.experimental.unary_unary(request, target, '/brain_ai.BrainAI/Health',
            brain__ai__pb2.HealthRequest.SerializeToString,
            brain__ai__pb2.HealthResponse.FromString,
            options, channel_credentials,
            insecure, call_credentials, compression, wait_for_ready, timeout, metadata)
```

```cpp
#include "brain_ai/workspace/quantum_strict.hpp"
#include <gtest/gtest.h>
#include <cmath>

using namespace brain_ai::workspace;

TEST(QuantumStrict, InitialStateValid) {
    QuantumConfig config;
    config.rng_seed = 123;
    QuantumStrict q(config);

    // Initial state is maximally mixed
    EXPECT_NEAR(q.trace_error(), 0.0, 1e-10);
    EXPECT_TRUE(q.is_hermitian());
    EXPECT_TRUE(q.is_positive_semidefinite());
    EXPECT_GE(q.min_eigenvalue(), -1e-10);
    EXPECT_NEAR(q.von_neumann_entropy(), std::log(7.0), 1e-8);
}

TEST(QuantumStrict, CPTPPreservation) {
    QuantumConfig config;
    config.dt = 1e-4;
    config.rng_seed = 456;
    QuantumStrict q(config);

    for (int step = 0; step < 100; ++step) {
        q.step();
        // CPTP invariants must hold at every step
        EXPECT_LT(q.trace_error(), 1e-8) << "Trace drift at step " << step;
        EXPECT_TRUE(q.is_hermitian()) << "Non-Hermitian at step " << step;
        EXPECT_TRUE(q.is_positive_semidefinite()) << "Negative eigenvalue at step " <<
step;

        double s = q.von_neumann_entropy();
        EXPECT_GE(s, -1e-10) << "Negative entropy at step " << step;
        EXPECT_LE(s, std::log(7.0) + 1e-8) << "Entropy exceeds maximum at step " << step;
    }
}

TEST(QuantumStrict, MeasurementCollapse) {
    QuantumConfig config;
    config.rng_seed = 321;
    QuantumStrict q(config);

    // Perform measurement
    q.measure_computational_basis();
    int state = q.measured_state();
    EXPECT_GE(state, 0);
    EXPECT_LT(state, 7);

    // After measurement, should be pure state
    EXPECT_NEAR(q.von_neumann_entropy(), 0.0, 1e-10);

    // Density matrix should be projector |k><k|
    auto rho = q.density_matrix();
    for (int i = 0; i < 7; ++i) {
        for (int j = 0; j < 7; ++j) {
            if (i == state && j == state) {
                EXPECT_NEAR(std::abs(rho(i, j)), 1.0, 1e-10);
            } else {
                EXPECT_NEAR(std::abs(rho(i, j)), 0.0, 1e-10);
            }
        }
    }
}
```

```
TEST(QuantumStrict, DeterministicReproducibility) {
    QuantumConfig config1;
    config1.rng_seed = 12345;
    QuantumStrict q1(config1);

    QuantumConfig config2;
    config2.rng_seed = 12345;
    QuantumStrict q2(config2);

    // Same seed should produce identical evolution
    for (int step = 0; step < 50; ++step) {
        q1.step();
        q2.step();
        auto rho1 = q1.density_matrix();
        auto rho2 = q2.density_matrix();
        double diff = (rho1 - rho2).norm();
        EXPECT_LT(diff, 1e-14) << "Non-deterministic at step " << step;
        EXPECT_DOUBLE_EQ(q1.von_neumann_entropy(), q2.von_neumann_entropy());
    }
}
```

```
{
    "allow": [
        {"op": "brain.step"},
        {"op": "brain.measure"},
        {"op": "memory.append"},
        {"op": "plan.explain"},
        {"op": "selfedit.propose"},
        {"op": "selfedit.approve"},
        {"op": "selfedit.apply"}
    ],
    "deny": [
        {"op": "system.exec"},
        {"op": "net.open"},
        {"op": "fs.write", "path": "/*", "except": ["/var/log/cortex/*.log"]}
    ],
    "quorum": 1
}
```

```
# Denylist of dangerous patterns to scan for in proposed code
DENY_PATTERNS = [
    "subprocess.Popen",
    "os.system",
    "open(", # Overly broad, but safe
    "socket.",
    "eval(",
    "exec(",
    "import os",
    "import subprocess"
]

def scan_text(txt: str) -> list[str]:
    """Scans text for denylisted patterns."""
    hits = [p for p in DENY_PATTERNS if p in txt]
    return list(set(hits))
```

```
#pragma once

#include "brain_ai.grpc.pb.h"
#include "brain_ai/workspace/quantum_strict.hpp"
#include "brain_ai/evolve/merkle_log.hpp"
#include <grpc++/grpc++.h>
#include <memory>
#include <thread>

namespace brain_ai {
namespace api {

// Implements the gRPC BrainAI service
class BrainServiceImpl final : public BrainAI::Service {
public:
    BrainServiceImpl(
        const std::string& audit_log_path,
        const std::string& keys_path
    );

    grpc::Status Step(
        grpc::ServerContext* context,
        const StepRequest* request,
        StepResponse* response) override;

    grpc::Status GetState(
        grpc::ServerContext* context,
        const GetStateRequest* request,
        GetStateResponse* response) override;

    grpc::Status StreamInference(
        grpc::ServerContext* context,
        grpc::ServerReader<InferenceInput>* reader,
        grpc::ServerWriter<InferenceOutput>* writer) override;

    grpc::Status Health(
        grpc::ServerContext* context,
        const HealthRequest* request,
        HealthResponse* response) override;

private:
    std::unique_ptr<workspace::QuantumStrict> quantum_;
    std::unique_ptr<evolve::MerkleAuditLog> audit_log_;
    std::vector<uint8_t> signing_key_;
    std::vector<uint8_t> public_key_;
};

// Simple HTTP server for the /metrics endpoint
class MetricsServer {
public:
    explicit MetricsServer(int port);
    void start();
    void stop();

private:
    int port_;
    int server_fd_;
    std::thread server_thread_;
    std::atomic<bool> running_;

    void serve();
    std::string handle_request(const std::string& request);
};

} // namespace api
} // namespace brain_ai
```

```cpp
#pragma once

#include <string>
#include <map>
#include <mutex>
#include <vector>
#include <limits>

namespace brain_ai {
namespace evolve {

enum class MetricType {
    COUNTER,
    GAUGE,
    HISTOGRAM
};

// Singleton class for exporting Prometheus metrics
class Metrics {
public:
    static Metrics& instance();

    void increment(const std::string& name, double value = 1.0);
    void set_gauge(const std::string& name, double value);
    void observe_histogram(const std::string& name, double value);

    // Export in Prometheus text format
    std::string export_prometheus() const;

    // Register metric with metadata
    void register_metric(
        const std::string& name,
        MetricType type,
        const std::string& help
    );

private:
    Metrics() = default;

    struct MetricData {
        MetricType type;
        std::string help;
        double value = 0.0;
        std::vector<double> histogram_buckets;
        std::vector<uint64_t> histogram_counts;
    };

    mutable std::mutex mutex_;
    std::map<std::string, MetricData> metrics_;
};

} // namespace evolve
} // namespace brain_ai
```

```
import json
import hashlib
import hmac
import pathlib
import nacl.signing
import nacl.encoding
from jsonschema import validate
from .schema import MANIFEST_SCHEMA

class Signer:
    """
    Handles Ed25519 signing and verification for manifests.
    This should wrap a real KMS/HSM in production.
    """
    def __init__(self, key_ref: str):
        # key_ref would be "aws-kms:alias/cortex-approve"
        # For demo, we'll use a local file.
        self.key_ref = key_ref
        self.sk_file = pathlib.Path("config/editor_sk.key")
        self.pk_file = pathlib.Path("config/editor_pk.key")

        if not self.sk_file.exists():
            self.sk = nacl.signing.SigningKey.generate()
            self.sk_file.write_bytes(self.sk.encode())
            self.pk_file.write_bytes(self.sk.verify_key.encode())
        else:
            self.sk = nacl.signing.SigningKey(self.sk_file.read_bytes())

        self.pk = self.sk.verify_key
        print(f"Self-Editor public key:
{self.pk.encode(nacl.encoding.HexEncoder).decode()}")

    def sign(self, msg: bytes) -> bytes:
        # In prod, this calls:
        # response = kms.sign(KeyId=self.key_ref, Message=msg, ...)
        # return response['Signature']
        return self.sk.sign(msg).signature

    def verify(self, msg: bytes, sig: bytes, pub_key_bytes: bytes = None) -> bool:
        try:
            pk_to_use = nacl.signing.VerifyKey(pub_key_bytes) if pub_key_bytes else
self.pk
            pk_to_use.verify(msg, sig)
            return True
        except Exception:
            return False

# --- Manifest Logic ---

def normalize(m: dict) -> bytes:
    """Serializes a manifest into a canonical JSON string for signing."""
    return json.dumps(m, separators=(",", ":"), sort_keys=True).encode()

def digest(m: dict) -> str:
    """Computes the SHA-256 digest of the canonical manifest."""
    return hashlib.sha256(normalize(m)).hexdigest()

def validate_manifest(m: dict, max_bytes: int) -> int:
    """Validates the manifest against the schema and size limits."""
    validate(m, MANIFEST_SCHEMA)
    total_size = sum(len(e.get("text", "").encode()) for e in m["edits"])
    if total_size > max_bytes:
        raise ValueError(f"Edit size {total_size} exceeds max {max_bytes}")
    return total_size

def verify_hmac(m: dict, sig: str, token: str) -> bool:
```

```
"""DEPRECATED: Verifies an HMAC signature."""
expected_sig = hmac.new(token.encode(), normalize(m), hashlib.sha256).hexdigest()
return hmac.compare_digest(expected_sig, sig)
```

```
import os
import json
import pathlib
import numpy as np
from .workspace_adapter import embed_for_vdb

try:
    import faiss
    HAVE_FAISS = True
    print("FAISS imported successfully.")
except ImportError:
    HAVE_FAISS = False
    print("FAISS not found, falling back to numpy cosine similarity.")

class VectorMemory:
    """
    Manages vector embeddings for episodic memory retrieval.
    Uses FAISS if available, otherwise falls back to pure numpy.
    """
    def __init__(self, jsonl_path: str, index_path: str, dim: int = 384):
        self.log_path = pathlib.Path(jsonl_path)
        self.index_path = index_path
        self.dim = dim
        self.vecs = []
        self.meta = []
        self.index = None
        self._load_from_log()

    def _load_from_log(self):
        """Load existing vectors and metadata from the JSONL log."""
        if not self.log_path.exists():
            return

        with self.log_path.open("r") as f:
            for line in f:
                try:
                    entry = json.loads(line)
                    if "text" in entry:
                        v = embed_for_vdb(entry["text"], self.dim)
                        self.vecs.append(v)
                        self.meta.append(entry)
                except json.JSONDecodeError:
                    continue # Skip corrupted lines

        if self.vecs:
            self.rebuild_index()

    def rebuild_index(self):
        """Rebuilds the FAISS index from in-memory vectors."""
        if HAVE_FAISS:
            self.index = faiss.IndexFlatIP(self.dim) # IP = Inner Product (Cosine)
            vecs_np = np.array(self.vecs).astype(np.float32)
            if vecs_np.shape[0] > 0:
                self.index.add(vecs_np)
                faiss.write_index(self.index, self.index_path)
        else:
            self.vecs_np = np.array(self.vecs).astype(np.float32)


    def add(self, text: str, meta: dict):
        """Adds a new text and metadata to the memory."""
        v = embed_for_vdb(text, self.dim)
        self.vecs.append(v)
        self.meta.append(meta)

        # Append to log
```

```
    with self.log_path.open("a") as f:
        f.write(json.dumps({"text": text, **meta}) + "\n")

    # Update index
    if HAVE_FAISS:
        if self.index is None:
            self.index = faiss.IndexFlatIP(self.dim)
        self.index.add(v.reshape(1, -1))
        faiss.write_index(self.index, self.index_path)
    else:
        self.vecs_np = np.array(self.vecs).astype(np.float32)

def search(self, query: str, top_k: int = 5) -> list[dict]:
    """Searches the memory for the most relevant entries."""
    if not self.vecs:
        return []

    q = embed_for_vdb(query, self.dim).reshape(1, -1)

    if HAVE_FAISS and self.index is not None:
        D, I = self.index.search(q, top_k)
        hits = []
        for idx, score in zip(I[0], D[0]):
            if 0 <= idx < len(self.meta):
                hits.append({"score": float(score), **self.meta[idx]})
        return hits

    # Fallback to numpy
    if hasattr(self, 'vecs_np'):
        scores = (q @ self.vecs_np.T)[0]
        order = np.argsort(scores)[::-1][:top_k]
        return [{"score": scores[i], **self.meta[i]} for i in order]

    return []
```

```
cmake_minimum_required(VERSION 3.20)
project(brain-ai-production VERSION 2.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_POSITION_INDEPENDENT_CODE ON)

# Options
option(ENABLE_TESTS "Build tests" ON)
option(ENABLE_BENCHMARKS "Build benchmarks" ON)
option(USE_SANITIZERS "Enable ASan/UBSan" OFF)

# Dependencies
find_package(Eigen3 3.4 REQUIRED)
find_package(PkgConfig REQUIRED)
pkg_check_modules(SODIUM REQUIRED libsodium)
find_package(gRPC REQUIRED)
find_package(Protobuf REQUIRED)

# GTest via FetchContent
include(FetchContent)

if(ENABLE_TESTS)
    FetchContent_Declare(
        googletest
        GIT_REPOSITORY https://github.com/google/googletest.git
        GIT_TAG v1.14.0
    )
    FetchContent_MakeAvailable(googletest)
endif()

if(ENABLE_BENCHMARKS)
    FetchContent_Declare(
        benchmark
        GIT_REPOSITORY https://github.com/google/benchmark.git
        GIT_TAG v1.8.3
    )
    FetchContent_MakeAvailable(benchmark)
endif()

# Protobuf & gRPC
get_target_property(GRPC_CPP_PLUGIN gRPC::grpc_cpp_plugin LOCATION)
get_target_property(PROTOC_EXE Protobuf::protoc LOCATION)

set(PROTO_FILES ${CMAKE_CURRENT_SOURCE_DIR}/proto/brain_ai.proto)
set(PROTO_GEN_DIR ${CMAKE_CURRENT_BINARY_DIR}/generated)

add_custom_command(
    OUTPUT ${PROTO_GEN_DIR}/brain_ai.pb.cc ${PROTO_GEN_DIR}/brain_ai.pb.h
${PROTO_GEN_DIR}/brain_ai.grpc.pb.cc ${PROTO_GEN_DIR}/brain_ai.grpc.pb.h
    COMMAND ${PROTOC_EXE}
    ARGS --grpc_out=${PROTO_GEN_DIR}
         --cpp_out=${PROTO_GEN_DIR}
         -I ${CMAKE_CURRENT_SOURCE_DIR}/proto
         --plugin=protoc-gen-grpc=${GRPC_CPP_PLUGIN}
         ${PROTO_FILES}
    DEPENDS ${PROTO_FILES}
)
add_custom_target(GenProto DEPENDS ${PROTO_GEN_DIR}/brain_ai.pb.cc
${PROTO_GEN_DIR}/brain_ai.pb.h ${PROTO_GEN_DIR}/brain_ai.grpc.pb.cc
${PROTO_GEN_DIR}/brain_ai.grpc.pb.h)

# Add generated files to include path
include_directories(${PROTO_GEN_DIR})

# Library
```

```
add_library(brain_ai
    src/workspace/quantum_strict.cpp
    src/evolve/merkle_log.cpp
    src/evolve/sandbox_runner.cpp
    src/evolve/metrics.cpp
    src/api/brain_service.cpp
    src/api/metrics_server.cpp
    ${PROTO_GEN_DIR}/brain_ai.pb.cc
    ${PROTO_GEN_DIR}/brain_ai.grpc.pb.cc
)
add_dependencies(brain_ai GenProto)

target_include_directories(brain_ai PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
)

target_link_libraries(brain_ai
    PUBLIC
        Eigen3::Eigen
        gRPC::grpc++
        Protobuf::libprotobuf
    PRIVATE
        ${SODIUM_LIBRARIES}
        seccomp
        Threads::Threads
)

target_compile_definitions(brain_ai PRIVATE HAVE_SODIUM)
set_target_properties(brain_ai PROPERTIES
    VERSION ${PROJECT_VERSION}
    SOVERSION 2
)

# Main Executable
add_executable(brain_ai_server src/main.cpp)
add_dependencies(brain_ai_server GenProto)
target_link_libraries(brain_ai_server brain_ai)

# Compiler flags
add_compile_options(-Wall -Wextra -Werror -fvisibility=hidden)
if(USE_SANITIZERS)
    add_compile_options(-fsanitize=address,undefined -fno-omit-frame-pointer)
    add_link_options(-fsanitize=address,undefined)
endif()

# Tests
if(ENABLE_TESTS)
    enable_testing()

    add_executable(test_quantum tests/test_quantum_strict.cpp)
    add_dependencies(test_quantum GenProto)
    target_link_libraries(test_quantum brain_ai GTest::gtest_main)
    add_test(NAME quantum_cptp COMMAND test_quantum)

    add_executable(test_merkle tests/test_merkle_log.cpp)
    add_dependencies(test_merkle GenProto)
    target_link_libraries(test_merkle brain_ai GTest::gtest_main)
    add_test(NAME merkle_chain COMMAND test_merkle)
endif()

# Install
install(TARGETS brain_ai_server
    RUNTIME DESTINATION bin
)
install(TARGETS brain_ai
```

```
    EXPORT brain_ai-targets
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    RUNTIME DESTINATION bin
)
install(DIRECTORY include/ DESTINATION include)
```

```cpp
#include "brain_ai/evolve/merkle_log.hpp"
#include <sodium.h>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <chrono>
#include <stdexcept>

// For file operations
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h> // for open()

using namespace brain_ai::evolve;

// Helper to ensure libsodium is initialized once
static void ensure_sodium_init() {
    static bool initialized = []() {
        if (sodium_init() < 0) {
            throw std::runtime_error("libsodium init failed");
        }
        return true;
    }();
    (void)initialized; // Suppress unused variable warning
}

std::vector<uint8_t> brain_ai::evolve::sha256(const std::vector<uint8_t>& data) {
    ensure_sodium_init();
    std::vector<uint8_t> hash(crypto_hash_sha256_BYTES);
    crypto_hash_sha256(hash.data(), data.data(), data.size());
    return hash;
}

std::string brain_ai::evolve::sha256_hex(const std::string& data) {
    std::vector<uint8_t> bytes(data.begin(), data.end());
    auto hash = sha256(bytes);
    std::ostringstream oss;
    for (uint8_t b : hash) {
        oss << std::hex << std::setw(2) << std::setfill('0') << (int)b;
    }
    return oss.str();
}

std::vector<uint8_t> brain_ai::evolve::ed25519_sign(
    const std::vector<uint8_t>& message,
    const std::vector<uint8_t>& secret_key) {

    ensure_sodium_init();
    if (secret_key.size() != crypto_sign_SECRETKEYBYTES) {
        throw std::invalid_argument("Invalid secret key size");
    }
    std::vector<uint8_t> sig(crypto_sign_BYTES);
    crypto_sign_detached(sig.data(), nullptr,
                         message.data(), message.size(),
                         secret_key.data());
    return sig;
}

bool brain_ai::evolve::ed25519_verify(
    const std::vector<uint8_t>& message,
    const std::vector<uint8_t>& signature,
    const std::vector<uint8_t>& public_key) {

    ensure_sodium_init();
    if (signature.size() != crypto_sign_BYTES ||
        public_key.size() != crypto_sign_PUBLICKEYBYTES) {
```

```
        return false;
    }
    return crypto_sign_verify_detached(signature.data(),
                                       message.data(), message.size(),
                                       public_key.data()) == 0;
}

std::pair<std::vector<uint8_t>, std::vector<uint8_t>> brain_ai::evolve::ed25519_keypair()
{
    ensure_sodium_init();
    std::vector<uint8_t> pk(crypto_sign_PUBLICKEYBYTES);
    std::vector<uint8_t> sk(crypto_sign_SECRETKEYBYTES);
    crypto_sign_keypair(pk.data(), sk.data());
    return {pk, sk};
}

void brain_ai::evolve::save_keypair(
    const std::string& dir,
    const std::vector<uint8_t>& pk,
    const std::vector<uint8_t>& sk) {

    // Create directory with restricted permissions
    mkdir(dir.c_str(), 0700);

    // Save public key (world-readable)
    std::ofstream pk_file(dir + "/public.key", std::ios::binary);
    pk_file.write(reinterpret_cast<const char*>(pk.data()), pk.size());
    pk_file.close();
    chmod((dir + "/public.key").c_str(), 0644);

    // Save secret key (owner-only)
    std::ofstream sk_file(dir + "/secret.key", std::ios::binary);
    sk_file.write(reinterpret_cast<const char*>(sk.data()), sk.size());
    sk_file.close();
    chmod((dir + "/secret.key").c_str(), 0600);
}

std::pair<std::vector<uint8_t>, std::vector<uint8_t>>
brain_ai::evolve::load_keypair(const std::string& dir) {
    std::vector<uint8_t> pk(crypto_sign_PUBLICKEYBYTES);
    std::vector<uint8_t> sk(crypto_sign_SECRETKEYBYTES);

    std::ifstream pk_file(dir + "/public.key", std::ios::binary);
    if (!pk_file) throw std::runtime_error("Failed to open public key");
    pk_file.read(reinterpret_cast<char*>(pk.data()), pk.size());

    std::ifstream sk_file(dir + "/secret.key", std::ios::binary);
    if (!sk_file) throw std::runtime_error("Failed to open secret key");
    sk_file.read(reinterpret_cast<char*>(sk.data()), sk.size());

    return {pk, sk};
}

MerkleAuditLog::MerkleAuditLog(const std::string& filepath, const std::string&
public_key_path)
    : filepath_(filepath), public_key_path_(public_key_path), sync_on_write_(true) {
    ensure_sodium_init();
}

void MerkleAuditLog::fsync_file() {
    if (!sync_on_write_) return;
    int fd = open(filepath_.c_str(), O_RDONLY);
    if (fd >= 0) {
        fsync(fd);
        close(fd);
    }
```

```
}

AuditEntry MerkleAuditLog::append(
    const std::string& event_type,
    const std::string& payload_json,
    const std::vector<uint8_t>& signing_key) {

    auto entries = read_all();
    uint64_t next_index = entries.empty() ? 0 : entries.back().index + 1;

    // Timestamp
    auto now = std::chrono::system_clock::now();
    auto time_t_now = std::chrono::system_clock::to_time_t(now);
    std::ostringstream oss;
    oss << std::put_time(std::gmtime(&time_t_now), "%Y-%m-%dT%H:%M:%SZ");

    AuditEntry entry;
    entry.index = next_index;
    entry.timestamp_iso8601 = oss.str();
    entry.event_type = event_type;
    entry.payload_json = payload_json;
    entry.previous_hash = entries.empty() ? std::vector<uint8_t>(32, 0) :
entries.back().current_hash;

    // Compute hash
    std::string data_to_hash = std::to_string(entry.index) +
                               entry.timestamp_iso8601 +
                               event_type + payload_json;

    std::vector<uint8_t> data_bytes(data_to_hash.begin(), data_to_hash.end());
    data_bytes.insert(data_bytes.end(), entry.previous_hash.begin(),
entry.previous_hash.end());

    entry.current_hash = sha256(data_bytes);

    // Sign
    entry.signature = ed25519_sign(data_bytes, signing_key);
    entry.merkle_proof = entry.current_hash; // Simplified

    // Append to file
    std::ofstream out(filepath_, std::ios::app | std::ios::binary);

    // Binary format: [index:8] [timestamp_len:4] [timestamp] [type_len:4] [type]
    // [payload_len:4] [payload] [prev_hash:32] [curr_hash:32] [sig:64]

    out.write(reinterpret_cast<const char*>(&entry.index), 8);

    uint32_t ts_len = entry.timestamp_iso8601.size();
    out.write(reinterpret_cast<const char*>(&ts_len), 4);
    out.write(entry.timestamp_iso8601.data(), ts_len);

    uint32_t type_len = entry.event_type.size();
    out.write(reinterpret_cast<const char*>(&type_len), 4);
    out.write(entry.event_type.data(), type_len);

    uint32_t payload_len = entry.payload_json.size();
    out.write(reinterpret_cast<const char*>(&payload_len), 4);
    out.write(entry.payload_json.data(), payload_len);

    out.write(reinterpret_cast<const char*>(entry.previous_hash.data()), 32);
    out.write(reinterpret_cast<const char*>(entry.current_hash.data()), 32);
    out.write(reinterpret_cast<const char*>(entry.signature.data()), 64);

    out.close();
    fsync_file();
    return entry;
```

```
}

std::vector<AuditEntry> MerkleAuditLog::read_all() const {
    std::vector<AuditEntry> entries;
    std::ifstream in(filepath_, std::ios::binary);
    if (!in) return entries;

    while (in.peek() != EOF) {
        AuditEntry entry;
        in.read(reinterpret_cast<char*>(&entry.index), 8);
        if (in.eof()) break;

        uint32_t ts_len;
        in.read(reinterpret_cast<char*>(&ts_len), 4);
        entry.timestamp_iso8601.resize(ts_len);
        in.read(&entry.timestamp_iso8601[0], ts_len);

        uint32_t type_len;
        in.read(reinterpret_cast<char*>(&type_len), 4);
        entry.event_type.resize(type_len);
        in.read(&entry.event_type[0], type_len);

        uint32_t payload_len;
        in.read(reinterpret_cast<char*>(&payload_len), 4);
        entry.payload_json.resize(payload_len);
        in.read(&entry.payload_json[0], payload_len);

        entry.previous_hash.resize(32);
        in.read(reinterpret_cast<char*>(entry.previous_hash.data()), 32);

        entry.current_hash.resize(32);
        in.read(reinterpret_cast<char*>(entry.current_hash.data()), 32);

        entry.signature.resize(64);
        in.read(reinterpret_cast<char*>(entry.signature.data()), 64);

        entries.push_back(entry);
    }
    return entries;
}

bool MerkleAuditLog::verify_chain(const std::vector<uint8_t>& public_key) const {
    auto entries = read_all();
    std::vector<uint8_t> expected_prev(32, 0);

    for (const auto& entry : entries) {
        // Verify hash chain
        if (entry.previous_hash != expected_prev) return false;

        // Recompute hash
        std::string data_to_hash = std::to_string(entry.index) +
                                   entry.timestamp_iso8601 +
                                   entry.event_type +
                                   entry.payload_json;

        std::vector<uint8_t> data_bytes(data_to_hash.begin(), data_to_hash.end());
        data_bytes.insert(data_bytes.end(), entry.previous_hash.begin(),
entry.previous_hash.end());

        auto computed = sha256(data_bytes);
        if (computed != entry.current_hash) return false;

        // Verify signature
        if (!ed25519_verify(data_bytes, entry.signature, public_key)) {
            return false;
        }
```

```
        expected_prev = entry.current_hash;
    }
    return true;
}

std::vector<uint8_t> MerkleAuditLog::root_hash() const {
    auto entries = read_all();
    return entries.empty() ? std::vector<uint8_t>(32, 0) : entries.back().current_hash;
}

void MerkleAuditLog::rotate_if_needed(size_t max_size_bytes) {
    struct stat st;
    if (stat(filepath_.c_str(), &st) == 0) {
        if ((size_t)st.st_size > max_size_bytes) {
            auto timestamp = std::chrono::system_clock::now().time_since_epoch().count();
            std::string archive_filepath = filepath_ + "." + std::to_string(timestamp);
            rename(filepath_.c_str(), archive_filepath.c_str());
        }
    }
}
```

```
import json
import hmac
import hashlib
import time
import pathlib
from fnmatch import fnmatch

class PolicyVM:
    """
    Verifies that all actions proposed by the Cortex
    are compliant with the rules defined in policy_rules.json.
    """
    def __init__(self, rules_path: str, require_signoff: bool = True):
        self.rules = json.loads(pathlib.Path(rules_path).read_text())
        self.require_signoff = require_signoff
        self.allow_ops = {r['op'] for r in self.rules.get('allow', [])}
        self.deny_rules = self.rules.get('deny', [])

    def verify(self, plan: dict) -> tuple[bool, str]:
        op = plan.get("op", "")

        # Deny first
        for rule in self.deny_rules:
            if rule["op"] == op:
                # Check path globs if they exist
                if "path" in rule:
                    plan_path = plan.get("path", "")
                    exceptions = rule.get("except", [])
                    if any(fnmatch(plan_path, g) for g in exceptions):
                        continue # This specific path is allowed
                    if fnmatch(plan_path, rule["path"]):
                        return False, f"denied: {op} on {plan_path}"
                else:
                    return False, f"denied: {op}"

        # Allowlist
        if op in self.allow_ops:
            return True, ""

        return False, f"not-allowed: {op}"

    def sign(self, plan: dict, token: str) -> str:
        """DEPRECATED: Use Ed25519 signer. Kept for reference."""
        msg = json.dumps(plan, sort_keys=True).encode()
        sig = hmac.new(token.encode(), msg, hashlib.sha256).hexdigest()
        return sig

    def validate_quorum(self) -> bool:
        # Placeholder for real quorum logic
        return self.rules.get("quorum", 1) <= 1
```

```
import os
import signal
import pathlib
import time

def armed(path: str) -> bool:
    """Checks if the kill-switch file exists."""
    return pathlib.Path(path).exists()

def arm(path: str):
    """Creates the kill-switch file."""
    pathlib.Path(path).touch()

def disarm(path: str):
    """Removes the kill-switch file."""
    pathlib.Path(path).unlink(missing_ok=True)

def enforce(path: str):
    """
    If the kill-switch is armed, terminates the entire process group
    and raises SystemExit.
    """
    if armed(path):
        print("KILL-SWITCH ENGAGED. TERMINATING.", flush=True)
        try:
            # Terminate the entire process group
            os.killpg(0, signal.SIGTERM)
        except Exception as e:
            print(f"Error during kill: {e}", flush=True)

        time.sleep(1) # Give time for signal to propagate
        raise SystemExit("Kill-switch engaged")
```

```
# Main configuration for the Python Cortex
[server]
host = "0.0.0.0"
port = 8080
# auth_token_file = "config/secrets.token" # Replaced by mTLS

[brain]
# Address of the C++ Brain Kernel gRPC server
address = "brain.svc:50051" # Use service name for mTLS
tls_ca_cert = "/etc/cortex/certs/ca.crt"
tls_client_cert = "/etc/cortex/certs/cortex-client.crt"
tls_client_key = "/etc/cortex/certs/cortex-client.key"

[limits]
max_plan_tokens = 1024
max_ws_len = 60
max_req_per_min = 120

[security]
require_policy_signoff = true
killswitch_file = "/tmp/brain.KILL"

[llm]
provider = "openai" # "openai", "anthropic", "openrouter"
model = "gpt-4o-mini"
timeout_s = 30
json_mode = true

[memory]
path = "memory_episodic.jsonl"
index = "memory_index.faiss"
top_k = 5

[selfedit]
root = "/srv/cortex_repo" # Writable git checkout of the cortex repo
allow_globs = ["src/**/*.py", "config/**/*.toml"]
deny_globs = ["**/.git/**", "**/secrets/**", "**/*.key", "**/*.pem"]
max_bytes = 200000
require_quorum = true

[tls]
enabled = true
ca = "/etc/cortex/certs/ca.crt"
server_cert = "/etc/cortex/certs/cortex.crt"
server_key = "/etc/cortex/certs/cortex.key"
require_client_cert = true

[auth]
bearer_fallback = false # Disable token auth in favor of mTLS

[signing]
kms_key_ref = "aws-kms:alias/cortex-approve" # Example KMS key
accept_pub_keys = ["pub_v3.pem", "pub_v2.pem"] # For key rotation
```

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: brain_ai.proto
"""Generated protocol buffer code."""
from google.protobuf.internal import enum_type_wrapper
from google.protobuf import descriptor as _descriptor
from google.protobuf import descriptor_pool as _descriptor_pool
from google.protobuf import message as _message
from google.protobuf import reflection as _reflection
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()




DESCRIPTOR =
_descriptor_pool.Default().AddSerializedFile(b'\n\x0e\x62rain_ai.proto\x12\x08\x62rain_ai
\"<\n\x0bStepRequest\x12\x1a\n\x12global_workspace\x18\x01
\x03(\x01\x12\x11\n\x07measure\x18\x02
\x01(\x08\"[\n\x0cStepResponse\x12\x0f\n\x07\x65ntropy\x18\x01
\x01(\x01\x12\x17\n\x0fmeasured_state\x18\x02 \x01(\x05\x12\x13\n\x0btrace_error\x18\x03
\x01(\x01\x12\x10\n\x08is_valid\x18\x04
\x01(\x08\"\x11\n\x0fGetStateRequest\"p\n\x10GetStateResponse\x12\x1d\n\x15\x64\x65nsity_
matrix_real\x18\x01 \x03(\x01\x12\x1d\n\x15\x64\x65nsity_matrix_imag\x18\x02
\x03(\x01\x12\x0f\n\x07\x65ntropy\x18\x03 \x01(\x01\x12\x13\n\t\x64imension\x18\x04
\x01(\x05\":\n\x0eInferenceInput\x12\x10\n\x08\x66\x65\x61tures\x18\x01
\x03(\x01\x12\x16\n\nrequest_id\x18\x02
\x01(\t\"J\n\x0fInferenceOutput\x12\x0e\n\x06logits\x18\x01
\x03(\x01\x12\x16\n\nrequest_id\x18\x02 \x01(\t\x12\x13\n\x0blatency_us\x18\x03
\x01(\x04\"\x0f\n\rHealthRequest\"\xa9\x01\n\x0eHealthResponse\x12.\n\x06status\x18\x01
\x01(\x0e\x32\x1e.brain_ai.HealthResponse.Status\x12\x0f\n\x07version\x18\x02
\x01(\t\x12\x16\n\x0euptime_seconds\x18\x03
\x01(\x04\"D\n\x06Status\x12\x0b\n\x07UNKNOWN\x10\x00\x12\x0b\n\x07SERVING\x10\x01\x12\x0
f\n\x0bNOT_SERVING\x10\x02\x12\x0f\n\x0bSERVICE_OFF\x10\x03\x32\xf6\x01\n\x07\x42rainAI\x
12\x31\n\x04Step\x12\x15.brain_ai.StepRequest\x1a\x16.brain_ai.StepResponse\"\x00\x12=\n\
x08GetState\x12\x19.brain_ai.GetStateRequest\x1a\x1a.brain_ai.GetStateResponse\"\x00\x12G
\n\x0fStreamInference\x12\x18.brain_ai.InferenceInput\x1a\x19.brain_ai.InferenceOutput\"\
x00(\x01\x30\x01\x12\x36\n\x06Health\x12\x17.brain_ai.HealthRequest\x1a\x1b.brain_ai.Heal
thResponse\"\x00\x62\x06proto3')

_STEPREQUEST = DESCRIPTOR.message_types_by_name['StepRequest']
_STEPRESPONSE = DESCRIPTOR.message_types_by_name['StepResponse']
_GETSTATEREQUEST = DESCRIPTOR.message_types_by_name['GetStateRequest']
_GETSTATERESPONSE = DESCRIPTOR.message_types_by_name['GetStateResponse']
_INFERENCEINPUT = DESCRIPTOR.message_types_by_name['InferenceInput']
_INFERENCEOUTPUT = DESCRIPTOR.message_types_by_name['InferenceOutput']
_HEALTHREQUEST = DESCRIPTOR.message_types_by_name['HealthRequest']
_HEALTHRESPONSE = DESCRIPTOR.message_types_by_name['HealthResponse']
_HEALTHRESPONSE_STATUS = _HEALTHRESPONSE.enum_types_by_name['Status']
StepRequest = _reflection.GeneratedProtocolMessageType('StepRequest',
(_message.Message,), {
  'DESCRIPTOR' : _STEPREQUEST,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.StepRequest)
  })
_sym_db.RegisterMessage(StepRequest)

StepResponse = _reflection.GeneratedProtocolMessageType('StepResponse',
(_message.Message,), {
  'DESCRIPTOR' : _STEPRESPONSE,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.StepResponse)
  })
_sym_db.RegisterMessage(StepResponse)
```

```
GetStateRequest = _reflection.GeneratedProtocolMessageType('GetStateRequest',
(_message.Message,), {
  'DESCRIPTOR' : _GETSTATEREQUEST,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.GetStateRequest)
  })
_sym_db.RegisterMessage(GetStateRequest)

GetStateResponse = _reflection.GeneratedProtocolMessageType('GetStateResponse',
(_message.Message,), {
  'DESCRIPTOR' : _GETSTATERESPONSE,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.GetStateResponse)
  })
_sym_db.RegisterMessage(GetStateResponse)

InferenceInput = _reflection.GeneratedProtocolMessageType('InferenceInput',
(_message.Message,), {
  'DESCRIPTOR' : _INFERENCEINPUT,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.InferenceInput)
  })
_sym_db.RegisterMessage(InferenceInput)

InferenceOutput = _reflection.GeneratedProtocolMessageType('InferenceOutput',
(_message.Message,), {
  'DESCRIPTOR' : _INFERENCEOUTPUT,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.InferenceOutput)
  })
_sym_db.RegisterMessage(InferenceOutput)

HealthRequest = _reflection.GeneratedProtocolMessageType('HealthRequest',
(_message.Message,), {
  'DESCRIPTOR' : _HEALTHREQUEST,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.HealthRequest)
  })
_sym_db.RegisterMessage(HealthRequest)

HealthResponse = _reflection.GeneratedProtocolMessageType('HealthResponse',
(_message.Message,), {
  'DESCRIPTOR' : _HEALTHRESPONSE,
  '__module__' : 'brain_ai_pb2'
  # @@protoc_insertion_point(class_scope:brain_ai.HealthResponse)
  })
_sym_db.RegisterMessage(HealthResponse)

_BRAINAI = DESCRIPTOR.services_by_name['BrainAI']
if _descriptor._USE_C_DESCRIPTORS == False:

  DESCRIPTOR._options = None
  _STEPREQUEST._serialized_start=28
  _STEPREQUEST._serialized_end=88
  _STEPRESPONSE._serialized_start=90
  _STEPRESPONSE._serialized_end=181
  _GETSTATEREQUEST._serialized_start=183
  _GETSTATEREQUEST._serialized_end=200
  _GETSTATERESPONSE._serialized_start=202
  _GETSTATERESPONSE._serialized_end=314
  _INFERENCEINPUT._serialized_start=316
  _INFERENCEINPUT._serialized_end=374
  _INFERENCEOUTPUT._serialized_start=376
  _INFERENCEOUTPUT._serialized_end=450
  _HEALTHREQUEST._serialized_start=452
```

```
  _HEALTHREQUEST._serialized_end=467
  _HEALTHRESPONSE._serialized_start=470
  _HEALTHRESPONSE._serialized_end=639
  _HEALTHRESPONSE_STATUS._serialized_start=571
  _HEALTHRESPONSE_STATUS._serialized_end=639
  _BRAINAI._serialized_start=642
  _BRAINAI._serialized_end=888
# @@protoc_insertion_point(module_scope)
```

```
#!/bin/bash
#
# Generates a private CA and all necessary server/client certs for mTLS.
# Store these securely. Do not check keys into git.
#

set -e
mkdir -p certs
cd certs

# 1. Create Root CA
echo "--- Generating Root CA ---"
openssl genrsa -out ca.key 4096
openssl req -x509 -new -nodes -key ca.key -sha256 -days 1095 \
    -subj "/CN=FDQC Internal CA" -out ca.crt

# 2. Create Brain Kernel Server Cert
echo "--- Generating Brain Kernel Server Cert ---"
openssl genrsa -out brain.key 4096
openssl req -new -key brain.key -subj "/CN=brain.svc" -out brain.csr
printf
"subjectAltName=DNS:brain.svc,DNS:localhost,IP:127.0.0.1\nextendedKeyUsage=serverAuth\n"
> brain.ext
openssl x509 -req -in brain.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out brain.crt -days 365 -sha256 -extfile brain.ext

# 3. Create Cortex Server Cert
echo "--- Generating Cortex Server Cert ---"
openssl genrsa -out cortex.key 4096
openssl req -new -key cortex.key -subj "/CN=cortex.svc" -out cortex.csr
printf
"subjectAltName=DNS:cortex.svc,DNS:localhost,IP:127.0.0.1\nextendedKeyUsage=serverAuth\n"
> cortex.ext
openssl x509 -req -in cortex.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out cortex.crt -days 365 -sha256 -extfile cortex.ext

# 4. Create Cortex Client Cert (for talking to Brain)
echo "--- Generating Cortex Client Cert ---"
openssl genrsa -out cortex-client.key 4096
openssl req -new -key cortex-client.key -subj "/CN=cortex-client" -out cortex-client.csr
printf "extendedKeyUsage=clientAuth\n" > client.ext
openssl x509 -req -in cortex-client.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out cortex-client.crt -days 365 -sha256 -extfile client.ext

# 5. Create Operator Client Cert (for talking to Cortex)
echo "--- Generating Operator Client Cert ---"
openssl genrsa -out operator.key 4096
openssl req -new -key operator.key -subj "/CN=operator" -out operator.csr
openssl x509 -req -in operator.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out operator.crt -days 365 -sha256 -extfile client.ext

echo "--- Cleaning up CSRs and extensions ---"
rm -f *.csr *.ext *.srl
cd ..
echo "--- Done. Certs generated in ./certs/ ---"
```

```
#pragma once

#include <Eigen/Dense>
#include <complex>
#include <vector>
#include <random>

namespace brain_ai {
namespace workspace {

// Configuration for the quantum workspace
struct QuantumConfig {
    int dimension = 7;
    double dt = 1e-3; // Timestep
    double eigenvalue_floor = 1e-12;
    double trace_tolerance = 1e-10;
    double decoherence_rate = 1e-8;
    bool adaptive_step = true;
    double dt_min = 1e-6;
    double dt_max = 1e-2;
    uint64_t rng_seed = 42; // Deterministic by default
};

// Implements the FDQC n=7 quantum workspace
// Uses Complete Positive Trace-Preserving (CPTP) maps via Kraus operators
// and robust Eigen-based complex eigensolvers.
class QuantumStrict {
public:
    explicit QuantumStrict(const QuantumConfig& config = {});

    // Main evolution step
    void step();

    // Project from the 60D Global Workspace into the 7D quantum state
    void project_from_global_workspace(
        const std::vector<double>& gw,
        const std::vector<std::complex<double>>& learned_weights
    );

    // --- Observables ---
    double von_neumann_entropy() const { return entropy_; }
    int measured_state() const { return measured_state_; } // -1 = no measurement
    const Eigen::MatrixXcd& density_matrix() const { return rho_; }

    // --- Validation ---
    double trace_error() const { return std::abs(rho_.trace() - 1.0); }
    bool is_hermitian(double tol = 1e-10) const {
        return (rho_ - rho_.adjoint()).norm() < tol;
    }
    bool is_positive_semidefinite() const;
    double min_eigenvalue() const;

    // --- Measurement ---
    // Explicitly collapse the state via POVM
    void measure_computational_basis();

private:
    // Internal methods
    void evolve_cptp_kraus();
    void enforce_cptp_projection();
    double compute_entropy_exact() const;
    void adaptive_timestep();
    void update_kraus_operators(); // Pre-compute Kraus operators

    QuantumConfig config_;
    Eigen::MatrixXcd rho_; // The 7x7 density matrix
```

```
    Eigen::MatrixXcd H_;   // Hamiltonian
    std::vector<Eigen::MatrixXcd> L_; // Lindblad operators
    std::vector<Eigen::MatrixXcd> K_; // Cached Kraus operators

    double entropy_;
    int measured_state_;
    double current_dt_;
    std::mt19937_64 rng_; // Seeded, reproducible RNG
};

} // namespace workspace
} // namespace brain_ai
```

```yaml
name: CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        sanitizer: [none, asan, ubsan] # tsan is also possible

    steps:
    - uses: actions/checkout@v4
      with:
        submodules: 'recursive' # In case you add submodules

    - name: Install dependencies
      run: |
        sudo apt-get update
        sudo apt-get install -y \
          build-essential \
          cmake \
          ninja-build \
          pkg-config \
          libeigen3-dev \
          libsodium-dev \
          libseccomp-dev \
          libgrpc++-dev \
          libprotobuf-dev \
          protobuf-compiler-grpc

    - name: Configure
      run: |
        FLAGS=""
        if [ "${{ matrix.sanitizer }}" != "none" ]; then
          FLAGS="-DUSE_SANITIZERS=ON"
        fi
        cmake -B build -S . -GNinja $FLAGS

    - name: Build
      run: cmake --build build

    - name: Test
      run: |
        cd build
        ctest --output-on-failure --timeout 300

  security:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Run CodeQL
      uses: github/codeql-action/analyze@v3
```

```
# JSON Schema for a self-edit manifest
MANIFEST_SCHEMA = {
    "type": "object",
    "required": ["id", "targets", "edits", "tests", "reason"],
    "properties": {
        "id": {"type": "string", "pattern": "^SE-[0-9]+$"},
        "targets": {"type": "array", "items": {"type": "string"}},
        "edits": {
            "type": "array",
            "items": {
                "type": "object",
                "required": ["path", "op", "start", "end", "text"],
                "properties": {
                    "path": {"type": "string"},
                    "op": {"enum": ["replace", "insert", "delete"]},
                    "start": {"type": "integer", "minimum": 0},
                    "end": {"type": "integer", "minimum": 0},
                    "text": {"type": "string"}
                }
            }
        },
        "tests": {"type": "array", "items": {"type": "string"}},
        "reason": {"type": "string", "maxLength": 2000}
    }
}
```

```
from .diff_apply import run_sandboxed

def run_all(cwd: str) -> tuple[bool, dict]:
    """
    Runs the full test and static analysis suite inside the sandbox.
    """
    cmds = [
        ["pytest", "-q"],
        ["python3", "-m", "pyflakes", "."],
        # ["coverage", "run", "-m", "pytest", "-q"],
        # ["coverage", "report", "--fail-under=80"] # Coverage can be slow
    ]

    for c in cmds:
        print(f"Running test command: {' '.join(c)}")
        code, out, err = run_sandboxed(c, cwd)
        if code != 0:
            return False, {"cmd": c, "out": out, "err": err}

    return True, {}
```

```cpp
#include "brain_ai/api/brain_service.hpp"
#include <iostream>
#include <csignal>
#include <string>
#include <grpc++/grpc++.h>
#include <grpc++/ssl_server_credentials.h>
#include <fstream>

using namespace brain_ai;

static std::unique_ptr<grpc::Server> g_server;
static std::unique_ptr<api::MetricsServer> g_metrics;

void signal_handler(int signum) {
    std::cout << "Shutting down gracefully..." << std::endl;
    if (g_server) {
        g_server->Shutdown();
    }
    if (g_metrics) {
        g_metrics->stop();
    }
}

std::string load_file(const std::string& filepath) {
    std::ifstream file(filepath);
    if (!file.is_open()) {
        throw std::runtime_error("Failed to open file: " + filepath);
    }
    std::stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
}

int main(int argc, char** argv) {
    // Parse arguments (simplified)
    int grpc_port = 50051;
    int metrics_port = 9090;
    std::string audit_log = "brain_ai_audit.log";
    std::string keys_dir = "/etc/brain_ai/keys"; // Default production key path
    std::string certs_dir = "/etc/brain_ai/certs"; // Default production cert path
    bool use_mtls = true;

    // TODO: Add proper argument parsing
    std::cout << "Using keys from: " << keys_dir << std::endl;
    std::cout << "Using certs from: " << certs_dir << std::endl;

    // Setup signal handlers
    std::signal(SIGINT, signal_handler);
    std::signal(SIGTERM, signal_handler);

    try {
        // Start metrics server
        g_metrics = std::make_unique<api::MetricsServer>(metrics_port);
        g_metrics->start();
        std::cout << "Metrics server listening on port " << metrics_port << std::endl;

        // Start gRPC server
        api::BrainServiceImpl service(audit_log, keys_dir);

        grpc::ServerBuilder builder;
        std::shared_ptr<grpc::ServerCredentials> creds;

        if (use_mtls) {
            std::cout << "Starting gRPC server with mTLS" << std::endl;
            grpc::SslServerCredentialsOptions::PemKeyCertPair kp;
            kp.private_key = load_file(certs_dir + "/brain.key");
```

```
            kp.cert_chain = load_file(certs_dir + "/brain.crt");

            grpc::SslServerCredentialsOptions opts;
            opts.pem_key_cert_pairs.push_back(kp);
            opts.pem_root_certs = load_file(certs_dir + "/ca.crt");
            opts.client_certificate_request =
GRPC_SSL_REQUEST_AND_REQUIRE_CLIENT_CERTIFICATE_AND_VERIFY;

            creds = grpc::SslServerCredentials(opts);
        } else {
            std::cout << "Starting gRPC server with Insecure credentials" << std::endl;
            creds = grpc::InsecureServerCredentials();
        }

        builder.AddListeningPort("0.0.0.0:" + std::to_string(grpc_port), creds);
        builder.RegisterService(&service);

        g_server = builder.BuildAndStart();
        std::cout << "gRPC server listening on port " << grpc_port << std::endl;
        g_server->Wait();

    } catch (const std::exception& e) {
        std::cerr << "Fatal error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

```
import os
import json
import httpx

class LLMClient:
    """
    Client for various LLM APIs, supporting strict JSON mode.
    """
    def __init__(self, provider: str, model: str, timeout: float = 30, json_mode: bool =
True):
        self.provider = provider
        self.model = model
        self.timeout = timeout
        self.json_mode = json_mode
        self.api_key = self._get_api_key()
        self.base_url, self.headers = self._get_provider_config()

    def _get_api_key(self):
        env_map = {
            "openai": "OPENAI_API_KEY",
            "anthropic": "ANTHROPIC_API_KEY",
            "openrouter": "OPENROUTER_API_KEY",
        }
        key = os.environ.get(env_map.get(self.provider))
        if not key:
            raise ValueError(f"{env_map.get(self.provider)} not set")
        return key

    def _get_provider_config(self):
        if self.provider == "openai":
            return (
                "https://api.openai.com/v1/chat/completions",
                {"Authorization": f"Bearer {self.api_key}"},
            )
        if self.provider == "anthropic":
            return (
                "https://api.anthropic.com/v1/messages",
                {"x-api-key": self.api_key, "anthropic-version": "2023-06-01"},
            )
        if self.provider == "openrouter":
            return (
                "https://openrouter.ai/api/v1/chat/completions",
                {"Authorization": f"Bearer {self.api_key}"},
            )
        raise ValueError("Unsupported provider")

    async def complete(self, system: str, user: str) -> dict:
        data = self._build_payload(system, user)

        async with httpx.AsyncClient(timeout=self.timeout) as cli:
            r = await cli.post(self.base_url, headers=self.headers, json=data)
            r.raise_for_status()
            return self._parse_response(r.json())

    def _build_payload(self, system, user):
        if self.provider == "anthropic":
            return {
                "model": self.model,
                "max_tokens": 1024,
                "system": system,
                "messages": [{"role": "user", "content": user}],
            }

        # Default to OpenAI/OpenRouter format
        payload = {
            "model": self.model,
```

```
        "messages": [
            {"role": "system", "content": system},
            {"role": "user", "content": user},
        ],
        "temperature": 0.2,
    }
    if self.json_mode and self.provider != "anthropic":
        payload["response_format"] = {"type": "json_object"}
    return payload

def _parse_response(self, resp: dict) -> dict:
    if self.provider == "anthropic":
        txt = resp["content"][0]["text"]
    else:
        txt = resp["choices"][0]["message"]["content"]

    try:
        return json.loads(txt)
    except json.JSONDecodeError:
        print(f"LLM did not return valid JSON: {txt}")
        raise
```

```cpp
#include "brain_ai/evolve/sandbox_runner.hpp"

#include <sys/prctl.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <linux/seccomp.h>
#include <linux/filter.h>
#include <linux/audit.h>
#include <seccomp.h>
#include <unistd.h>
#include <fcntl.h>
#include <sched.h>
#include <signal.h>
#include <chrono>
#include <cstring>
#include <stdexcept>
#include <thread>
#include <iostream>
#include <fstream>
#include <sstream>

using namespace brain_ai::evolve;

// Syscall number lookup helper
static int get_syscall_nr(const std::string& name) {
    int nr = seccomp_syscall_resolve_name(name.c_str());
    if (nr == __NR_SCMP_ERROR) {
        throw std::runtime_error("Unknown syscall: " + name);
    }
    return nr;
}

SandboxRunner::SandboxRunner(const SandboxConfig& config)
    : config_(config) {}

void SandboxRunner::setup_seccomp_filter() {
    // Create seccomp context: default KILL
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);
    if (!ctx) {
        throw std::runtime_error("seccomp_init failed");
    }

    // Add allowed syscalls
    for (const auto& syscall_name : config_.allowed_syscalls) {
        int nr = get_syscall_nr(syscall_name);
        if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, nr, 0) < 0) {
            seccomp_release(ctx);
            throw std::runtime_error("Failed to add rule for: " + syscall_name);
        }
    }

    // Allow essential syscalls for C++ runtime
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, __NR_rt_sigaction, 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, __NR_rt_sigprocmask, 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, __NR_futex, 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, __NR_close, 0);

    // Load filter
    if (seccomp_load(ctx) < 0) {
        seccomp_release(ctx);
        throw std::runtime_error("seccomp_load failed");
    }
    seccomp_release(ctx);
}
```

```
void SandboxRunner::setup_namespaces() {
    // Create new namespaces for isolation
    int flags = CLONE_NEWNS | // Mount namespace
                CLONE_NEWPID | // PID namespace
                CLONE_NEWUTS | // Hostname namespace
                CLONE_NEWIPC;  // IPC namespace

    if (!config_.enable_network) {
        flags |= CLONE_NEWNET;
    }
    if (unshare(flags) < 0) {
        throw std::runtime_error("unshare failed");
    }
}

void SandboxRunner::setup_cgroups(pid_t pid) {
    // Create cgroup for resource limits
    std::string cgroup_path = "/sys/fs/cgroup/brain_ai_sandbox_" + std::to_string(pid);
    mkdir(cgroup_path.c_str(), 0755);

    try {
        // Memory limit
        std::string mem_limit_file = cgroup_path + "/memory.max";
        std::ofstream mem_limit(mem_limit_file);
        mem_limit << config_.max_memory_bytes;
        mem_limit.close();

        // CPU limit (in microseconds per period)
        std::string cpu_quota_file = cgroup_path + "/cpu.max";
        std::ofstream cpu_quota(cpu_quota_file);
        cpu_quota << (config_.max_cpu_time_ms * 1000) << " 100000"; // quota / period
        cpu_quota.close();

        // Add process to cgroup
        std::string procs_file = cgroup_path + "/cgroup.procs";
        std::ofstream procs(procs_file);
        procs << pid;
        procs.close();
    } catch (const std::exception& e) {
        std::cerr << "Warning: Failed to set cgroups. " << e.what() << std::endl;
        // Continue without cgroup limits if they fail (e.g., permissions)
    }
}

void SandboxRunner::setup_chroot() {
    // Change root to isolated directory
    if (chroot(config_.chroot_dir.c_str()) < 0) {
        throw std::runtime_error("chroot failed");
    }
    if (chdir("/") < 0) {
        throw std::runtime_error("chdir failed after chroot");
    }
}

void SandboxRunner::drop_privileges() {
    // Set no new privs to prevent privilege escalation
    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) < 0) {
        throw std::runtime_error("prctl(NO_NEW_PRIVS) failed");
    }

    // Drop to unprivileged user
    if (setgid(65534) < 0) { // nogroup
        throw std::runtime_error("setgid failed");
    }
    if (setuid(65534) < 0) { // nobody user
        throw std::runtime_error("setuid failed");
```

```
    }
}

SandboxResult SandboxRunner::execute(
    const std::string& executable_path,
    const std::vector<std::string>& args) {
    return execute_with_stdin(executable_path, "", args);
}

SandboxResult SandboxRunner::execute_with_stdin(
    const std::string& executable_path,
    const std::string& stdin_data,
    const std::vector<std::string>& args) {

    SandboxResult result = {0, "", "", 0, 0, false, false, ""};

    // Create pipes for stdout/stderr capture
    int stdout_pipe[2], stderr_pipe[2], stdin_pipe[2];
    if (pipe(stdout_pipe) < 0 || pipe(stderr_pipe) < 0 || pipe(stdin_pipe) < 0) {
        throw std::runtime_error("pipe creation failed");
    }

    auto start_time = std::chrono::steady_clock::now();
    pid_t pid = fork();

    if (pid < 0) {
        throw std::runtime_error("fork failed");
    }

    if (pid == 0) {
        // --- Child process ---
        try {
            // Setup I/O redirection
            dup2(stdin_pipe[0], STDIN_FILENO);
            dup2(stdout_pipe[1], STDOUT_FILENO);
            dup2(stderr_pipe[1], STDERR_FILENO);
            close(stdin_pipe[0]); close(stdin_pipe[1]);
            close(stdout_pipe[0]); close(stdout_pipe[1]);
            close(stderr_pipe[0]); close(stderr_pipe[1]);

            // Setup resource limits
            struct rlimit rl;
            rl.rlim_cur = rl.rlim_max = config_.max_memory_bytes;
            setrlimit(RLIMIT_AS, &rl);
            rl.rlim_cur = rl.rlim_max = config_.max_cpu_time_ms / 1000;
            setrlimit(RLIMIT_CPU, &rl);
            rl.rlim_cur = rl.rlim_max = 1024; // Max 1024 open files
            setrlimit(RLIMIT_NOFILE, &rl);

            // Setup namespaces and isolation
            setup_namespaces();
            drop_privileges();
            setup_seccomp_filter();

            // Execute target
            std::vector<char*> exec_args;
            exec_args.push_back(const_cast<char*>(executable_path.c_str()));
            for (const auto& arg : args) {
                exec_args.push_back(const_cast<char*>(arg.c_str()));
            }
            exec_args.push_back(nullptr);

            execv(executable_path.c_str(), exec_args.data());

            // If we get here, exec failed
            std::cerr << "execv failed: " << strerror(errno) << std::endl;
```

```
        _exit(127);

    } catch (const std::exception& e) {
        std::cerr << "Sandbox setup failed: " << e.what() << std::endl;
        _exit(126);
    }
}

// --- Parent process ---
close(stdin_pipe[0]);
close(stdout_pipe[1]);
close(stderr_pipe[1]);

// Setup cgroups for child
setup_cgroups(pid);

// Write stdin data
if (!stdin_data.empty()) {
    write(stdin_pipe[1], stdin_data.data(), stdin_data.size());
}
close(stdin_pipe[1]);

// Setup timeout thread
std::atomic<bool> timed_out = false;
std::thread timeout_thread([&]() {
    std::this_thread::sleep_for(std::chrono::milliseconds(config_.timeout_ms));
    timed_out = true;
    kill(pid, SIGKILL);
});

// Read output in non-blocking mode
fcntl(stdout_pipe[0], F_SETFL, O_NONBLOCK);
fcntl(stderr_pipe[0], F_SETFL, O_NONBLOCK);
char buffer[4096];

while (true) {
    ssize_t n = read(stdout_pipe[0], buffer, sizeof(buffer));
    if (n > 0) {
        result.stdout_output.append(buffer, n);
    }
    n = read(stderr_pipe[0], buffer, sizeof(buffer));
    if (n > 0) {
        result.stderr_output.append(buffer, n);
    }

    int status;
    pid_t w = waitpid(pid, &status, WNOHANG);
    if (w == pid) {
        if (WIFEXITED(status)) {
            result.exit_code = WEXITSTATUS(status);
        } else if (WIFSIGNALED(status)) {
            result.exit_code = -WTERMSIG(status);
            if (WTERMSIG(status) == SIGSYS) {
                result.killed_violation = true;
                result.violation_reason = "Seccomp violation";
            }
        }
        break;
    }

    if (timed_out) {
        result.killed_timeout = true;
        break;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}
```

```
    timeout_thread.detach();
    close(stdout_pipe[0]);
    close(stderr_pipe[0]);

    auto end_time = std::chrono::steady_clock::now();
    result.elapsed_ms = std::chrono::duration_cast<std::chrono::milliseconds>(
        end_time - start_time).count();

    return result;
}
```

```
import json
import time
import pathlib

LOG_PATH = pathlib.Path("memory_episodic.jsonl")

def append(event: dict):
    """
    Appends an event to the append-only episodic log.
    """
    event_with_ts = {"ts": int(time.time()), **event}

    if not LOG_PATH.exists():
        LOG_PATH.write_text("") # Create file if not exists

    with LOG_PATH.open("a") as f:
        f.write(json.dumps(event_with_ts) + "\n")
```

```
#pragma once

#include <string>
#include <vector>
#include <cstdint>
#include <memory>

namespace brain_ai {
namespace evolve {

// Represents one entry in the immutable audit log
struct AuditEntry {
    uint64_t index;
    std::string timestamp_iso8601;
    std::string event_type;
    std::string payload_json;
    std::vector<uint8_t> previous_hash;
    std::vector<uint8_t> current_hash;
    std::vector<uint8_t> signature;
    std::vector<uint8_t> merkle_proof; // For subtree verification
};

// Provides a cryptographically-signed, tamper-proof Merkle-chained audit log
// using libsodium for Ed25519 signatures and SHA-256 hashing.
class MerkleAuditLog {
public:
    explicit MerkleAuditLog(const std::string& filepath, const std::string&
public_key_path);

    // Appends a new event, signs it, and hashes it into the chain.
    AuditEntry append(
        const std::string& event_type,
        const std::string& payload_json,
        const std::vector<uint8_t>& signing_key
    );

    // Verifies the entire chain's integrity (hashes and signatures).
    bool verify_chain(const std::vector<uint8_t>& public_key) const;

    // Verifies a single entry (not implemented in provided file, but good practice)
    // bool verify_entry(uint64_t index, const std::vector<uint8_t>& public_key) const;

    // Reads all entries from the log file.
    std::vector<AuditEntry> read_all() const;

    // Gets the hash of the latest entry (the root hash).
    std::vector<uint8_t> root_hash() const;

    // Durability: fsync on append
    void set_sync_mode(bool enable) { sync_on_write_ = enable; }

    // Log rotation
    void rotate_if_needed(size_t max_size_bytes);

private:
    std::string filepath_;
    std::string public_key_path_;
    bool sync_on_write_ = true;

    void fsync_file();
    std::vector<uint8_t> compute_merkle_root(const std::vector<AuditEntry>& entries)
const;
};

// --- Cryptographic Primitives (libsodium wrappers) ---
```

```
std::vector<uint8_t> sha256(const std::vector<uint8_t>& data);
std::string sha256_hex(const std::string& data);

std::vector<uint8_t> ed25519_sign(
    const std::vector<uint8_t>& message,
    const std::vector<uint8_t>& secret_key
);

bool ed25519_verify(
    const std::vector<uint8_t>& message,
    const std::vector<uint8_t>& signature,
    const std::vector<uint8_t>& public_key
);

std::pair<std::vector<uint8_t>, std::vector<uint8_t>> ed25519_keypair();

// --- Key Management ---

void save_keypair(
    const std::string& dir,
    const std::vector<uint8_t>& pk,
    const std::vector<uint8_t>& sk
);

std::pair<std::vector<uint8_t>, std::vector<uint8_t>>
load_keypair(const std::string& dir);

} // namespace evolve
} // namespace brain_ai
```

```
syntax = "proto3";

package brain_ai;

// The main gRPC service for the Brain Kernel
service BrainAI {
    // Executes one step of the quantum evolution
    rpc Step(StepRequest) returns (StepResponse);

    // Gets the current state of the density matrix
    rpc GetState(GetStateRequest) returns (GetStateResponse);

    // Bidirectional streaming for inference (example)
    rpc StreamInference(stream InferenceInput) returns (stream InferenceOutput);

    // Health check
    rpc Health(HealthRequest) returns (HealthResponse);
}

message StepRequest {
    repeated double global_workspace = 1; // 60D vector
    bool measure = 2; // Trigger a measurement
}

message StepResponse {
    double entropy = 1;
    int32 measured_state = 2;
    double trace_error = 3;
    bool is_valid = 4;
}

message GetStateRequest {}

message GetStateResponse {
    repeated double density_matrix_real = 1;
    repeated double density_matrix_imag = 2;
    double entropy = 3;
    int32 dimension = 4;
}

message InferenceInput {
    repeated double features = 1;
    string request_id = 2;
}

message InferenceOutput {
    repeated double logits = 1;
    string request_id = 2;
    uint64 latency_us = 3;
}

message HealthRequest {}

message HealthResponse {
    enum Status {
        UNKNOWN = 0;
        SERVING = 1;
        NOT_SERVING = 2;
    }
    Status status = 1;
    string version = 2;
    uint64 uptime_seconds = 3;
}
```