```
    def all_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        yield from self.sentencepiece_tokens()
        yield from self.added_tokens()

    def __repr__(self) -> str:
         return f"<SentencePieceVocab with {self.vocab_size_base} base tokens and {len(self.added_tokens_list)}
added tokens>"



class LlamaHfVocab(Vocab):
    tokenizer_model = "llama"
    name = "hfft"

    def __init__(self, base_path: Path):
        fname_tokenizer = base_path / 'tokenizer.json'
        # if this fails, FileNotFoundError propagates to caller
        with open(fname_tokenizer, encoding='utf-8') as f:
            tokenizer_json = json.load(f)

        # pre-check so we know if we need transformers
        tokenizer_model: dict[str, Any] = tokenizer_json['model']
        is_llama3 = (
            tokenizer_model['type'] == 'BPE' and tokenizer_model.get('ignore_merges', False)
            and not tokenizer_model.get('byte_fallback', True)
        )
        if is_llama3:
            raise TypeError('Llama 3 must be converted with BpeVocab')

        if not is_llama3 and (
            tokenizer_model['type'] != 'BPE' or not tokenizer_model.get('byte_fallback', False)
            or tokenizer_json['decoder']['type'] != 'Sequence'
        ):
            raise FileNotFoundError('Cannot find Llama BPE tokenizer')

        try:
            from transformers import AutoTokenizer
        except ImportError as e:
            raise ImportError(
                "To use LlamaHfVocab, please install the `transformers` package. "
                "You can install it with `pip install transformers`."
            ) from e

        # Allow the tokenizer to default to slow or fast versions.
        # Explicitly set tokenizer to use local paths.
        self.tokenizer = AutoTokenizer.from_pretrained(
            base_path,
            cache_dir=base_path,
            local_files_only=True,
        )
        assert self.tokenizer.is_fast  # assume tokenizer.json is used

        # Initialize lists and dictionaries for added tokens
        self.added_tokens_list = []
        self.added_tokens_dict = dict()
        self.added_tokens_ids  = set()

        # Process added tokens
        for tok, tokidx in sorted(
            self.tokenizer.get_added_vocab().items(), key=lambda x: x[1]
        ):
            # Only consider added tokens that are not in the base vocabulary
            if tokidx >= self.tokenizer.vocab_size:
                self.added_tokens_list.append(tok)
                self.added_tokens_dict[tok] = tokidx
                self.added_tokens_ids.add(tokidx)

        # Store special tokens and their IDs
        self.specials = {
            tok: self.tokenizer.get_vocab()[tok]
            for tok in self.tokenizer.all_special_tokens
        }
        self.special_ids = set(self.tokenizer.all_special_ids)

        # Set vocabulary sizes
        self.vocab_size_base = self.tokenizer.vocab_size
        self.vocab_size      = self.vocab_size_base + len(self.added_tokens_list)

        self.fname_tokenizer = fname_tokenizer

    def hf_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        reverse_vocab = {
            id: encoded_tok for encoded_tok, id in self.tokenizer.get_vocab().items()
        }

        for token_id in range(self.vocab_size_base):
            # Skip processing added tokens here
            if token_id in self.added_tokens_ids:
                continue

            # Convert token text to bytes
            token_text = reverse_vocab[token_id].encode("utf-8")

            # Yield token text, score, and type
            yield token_text, self.get_token_score(token_id), self.get_token_type(
                token_id, token_text, self.special_ids  # Reuse already stored special IDs
            )

    def get_token_type(self, token_id: int, token_text: bytes, special_ids: set[int]) -> gguf.TokenType:
        # Special case for byte tokens
        if re.fullmatch(br"<0x[0-9A-Fa-f]{2}>", token_text):
            return gguf.TokenType.BYTE

        # Determine token type based on whether it's a special token
        return gguf.TokenType.CONTROL if token_id in special_ids else gguf.TokenType.NORMAL

    def get_token_score(self, token_id: int) -> float:
        # Placeholder for actual logic to determine the token's score
        # This needs to be implemented based on specific requirements
        return -1000.0  # Default score

    def added_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        for text in self.added_tokens_list:
            if text in self.specials:
                toktype = self.get_token_type(self.specials[text], b'', self.special_ids)
                score = self.get_token_score(self.specials[text])
            else:
                toktype = gguf.TokenType.USER_DEFINED
                score = -1000.0

            yield text.encode("utf-8"), score, toktype

    def has_newline_token(self):
        return "<0x0A>" in self.tokenizer.vocab or "\n" in self.tokenizer.vocab

    def all_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        yield from self.hf_tokens()
        yield from self.added_tokens()

    def __repr__(self) -> str:
        return f"<LlamaHfVocab with {self.vocab_size_base} base tokens and {len(self.added_tokens_list)} added
tokens>"

==== winnow2.py ====
#!/usr/bin/env python
# coding: utf-8

# **Winnow2 Algorithm**
#
# *Author: Tirtharaj Dash, BITS Pilani, Goa Campus ([Homepage](https://tirtharajdash.github.io))*

import pickle

import numpy as np
import pandas as pd

# from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

# Use seed for reproducibility of results
seedval = 2018
np.random.seed(seedval)


# function to load data file and return train, val, test split
def load_data(trainfile="train.csv", testfile="test.csv"):
    # load the data and prepare X and y
    Data = pd.read_csv("train.csv", header=None)
    print(
        "Dimension of Data ( Instances: ",
        Data.shape[0],
        ", Features: ",
        Data.shape[1] - 1,
        " )",
    )

    # X = Data[Data.columns[0:3800]] #only for Hide-and-Seek features to match ILP results
    X = Data.drop([Data.columns[-1]], axis=1)  # for random features
    y = Data[Data.columns[-1]]
    print("Dimension of X: ", X.shape)
    print("Dimension of y: ", y.shape)

    # prepare the training and validation set from X and y
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.20, random_state=42, stratify=y, shuffle=True
    )

    print("\nDimension of X_train: ", X_train.shape)
    print("Dimension of y_train: ", y_train.shape)
    print("Dimension of X_val: ", X_val.shape)
    print("Dimension of y_val: ", y_val.shape)

    # load the holdout/test set
    Data_test = pd.read_csv("test.csv", header=None)
    print(
        "\nDimension of Data_test ( Instances: ",
        Data_test.shape[0],
        ", Features: ",
        Data_test.shape[1] - 1,
        " )",
    )

    # X_test = Data_test[Data_test.columns[0:3800]]
    X_test = Data_test.drop([Data_test.columns[-1]], axis=1)  # for random features
    y_test = Data_test[Data_test.columns[-1]]
    print("Dimension of X: ", X_test.shape)
    print("Dimension of y: ", y_test.shape)

    return X_train, y_train, X_val, y_val, X_test, y_test


# function to calculate accuracy
def accuracy_score(y, y_pred):
    acc = np.mean(y == y_pred)
    return acc


# function to calculate netsum and prediction for one instance
def predictOne(W, X, thres):
    netsum = np.sum(W * X)  # net sum

    # threshold check
    if netsum >= thres:
        y_hat = 1
    else:
        y_hat = 0
    return y_hat



# function to calculate netsums and predictions for all instances
def predictAll(W, X, thres):
    NetSum = np.dot(X, W)
    Idx = np.where(NetSum >= thres)
    y_pred = np.zeros(X.shape[0])
    y_pred[Idx] = 1

    return y_pred



# function to compute and print the classification summary
def ComputePerf(W, X, y, thres, print_flag=False):
    y_pred = predictAll(W, X, thres)  # compute the prediction
    acc = accuracy_score(y, y_pred)  # compute accuracy

    # print the summary if printflag is True
    if print_flag:
        print("Accuracy:", acc)
        # print(confusion_matrix(y, y_pred))
        # print(classification_report(y, y_pred))

    return acc



# function to train Winnow2 using grid-search (optional)
def TrainWinnow2(
    X_train, y_train, X_val, y_val, params, max_epoch=10, patience=20, verbose=False
):
    n = X_train.shape[1]
    best_perf = 0.0  # best val set performance so far
    grid_space = len(params["Alpha"]) * len(params["Thres"])  # size of grid space
    grid_iter = 0  # grid search iterator

    # model dictionary
    model = {"W": [], "alpha": None, "thres": None, "train_perf": 0.0, "val_perf": 0.0}

    # grid search and training
    for alpha in params["Alpha"]:
        for thres in params["Thres"]:
            grid_iter += 1

            print("-----------------------------------------------------------------")
            print(
                "Trying::\t alpha:",
                alpha,
                "threshold:",
                thres,
                "\t(",
                grid_iter,
                "of",
                grid_space,
                ")",
            )
            print("-----------------------------------------------------------------")

            W = np.ones(n)  # winnow2 initialisation
            piter = 0  # patience iteration
            modelfound = False  # model found flag

            for epoch in range(0, max_epoch):
                # Winnow loop (computation and update) starts
                for i in range(1, X_train.shape[0]):
                    y_hat = predictOne(W, X_train.iloc[i, :], thres)

                    # Winnow prediction is a mismatch
                    if y_train.iloc[i] != y_hat:
                        # active attribute indices
                        Idx = np.where(X_train.iloc[i, :] == 1)

                        if y_hat == 0:
                            W[Idx] *= alpha  # netsum was too small (promotion step)

                        else:
                            W[Idx] /= alpha  # netsum was too high (demotion step)

                # compute performance on val set
                val_perf = ComputePerf(W, X_val, y_val, thres)

                if verbose:
                    train_perf = ComputePerf(W, X_train, y_train, thres)
                    print(
                        "[Epoch %d] train_perf: %6.4f \tval_perf: %6.4f"
                        % (epoch, train_perf, val_perf)
                    )

                # is it a better model
                if val_perf > best_perf:
                    best_perf = val_perf
                    piter = 0  # reset the patience count
                    modelfound = True  # model is found
                    train_perf = ComputePerf(W, X_train, y_train, thres)

                    # update the model with new params
                    model["W"] = W.copy()  # optima W
                    model["alpha"] = alpha  # optimal alpha
                    model["thres"] = thres  # optimal threshold
                    model["train_perf"] = train_perf  # training performance
                    model["val_perf"] = val_perf  # validation performance

                    print(
                        "[Selected] at epoch",
                        epoch,
                        "\ttrain_perf: %6.4f \tval_perf: %6.4f"
                        % (train_perf, val_perf),
                    )

                else:
                    piter = piter + 1

                if piter >= patience:
                    print("Stopping early after epoch", (epoch + 1))
                    break

            if not modelfound:
                print("No better model found.\n")
            else:
                print("Model found and saved.\n")

    return model



# main function
def main():
    # call function to get the splits
    X_train, y_train, X_val, y_val, X_test, y_test = load_data(
        trainfile="train.csv", testfile="test.csv"
    )

    # Winnow2 algorithm hyperparameters (will be decided based on a validation set)
    n = X_train.shape[1]  # number of features
    Thres = [n, n // 2]  # decision threshold
    Alpha = [2, 3, 4]  # multiplicative factor for weight (promotion or demotion)

    # algorithm param dict
    params = {"Thres": Thres, "Alpha": Alpha}

    # trainign hyperparameters
    patience = 20  # patience period for early-stopping # noqa
    max_epoch = 200  # maximum Winnow epochs

    # call for training and model selection
    model = TrainWinnow2(
        X_train,
        y_train,
        X_val,
        y_val,
        params=params,
        max_epoch=max_epoch,
        patience=20,
        verbose=False,
    )

    # Optimal hyperparameters for Winnow2 using validation set
    print("Best model:", model)

    # test the performance of model on test set
    acc = ComputePerf(model["W"], X_test, y_test, model["thres"])
    print("Independent test accuracy:", acc)
    # store the results
    with open("score.txt", "w") as fp:
        fp.write(str(acc) + "\n")
    fp.close()

    # save the trained model (dict)
    with open("model.pkl", "wb") as fp:
        pickle.dump(model, fp, pickle.HIGHEST_PROTOCOL)
    fp.close()

    # load and test the saved model
    # with open('model.pkl', 'rb') as fp:
    #    savedmodel = pickle.load(fp)
    # acc = ComputePerf(savedmodel['W'], X_test, y_test, savedmodel['thres'])
    # print(acc)
    # print(classification_report(y_test, predictAll(savedmodel['W'], X_test, savedmodel['thres'])))
    # fp.close()


# for analysis of the weights
# with open('model.pkl', 'rb') as fp:
#    savedmodel = pickle.load(fp)
# df = pd.DataFrame(savedmodel['W'])
# df.to_csv("weights.csv")



if __name__ == "__main__":
    main()

==== winnow2_predict.py ====
#!/usr/bin/env python
# coding: utf-8

# **Winnow2 Algorithm**
#
# *Author: Tirtharaj Dash, BITS Pilani, Goa Campus ([Homepage](https://tirtharajdash.github.io))*

import pickle

import pandas as pd
import winnow2
from sklearn.metrics import classification_report

with open("./Results/chess3/model.pkl", "rb") as fp:
    savedmodel = pickle.load(fp)

Data = pd.read_csv("test3.pos.csv", header=None)
X = Data.drop([Data.columns[-1]], axis=1)
y = Data[Data.columns[-1]]

acc = winnow2.ComputePerf(savedmodel["W"], X, y, savedmodel["thres"])
y_pred = winnow2.predictAll(savedmodel["W"], X, savedmodel["thres"])
print(classification_report(y, y_pred))
```

```
==== word_dict_gen.py ====
import os
import json_lines
import re
from keras.preprocessing.text import Tokenizer

# Contextual embeddeding of symbols
texts = []  # list of text samples
id_list = []
question_list = []
label_list = []
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
TEXT_DATA_DIR = os.path.abspath('.') + "/data/pararule"
# TEXT_DATA_DIR = "D:\\AllenAI\\20_newsgroup"
Str = '.jsonl'
CONTEXT_TEXTS = []
#test_str = 'test'
meta_str = 'meta'

for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            fpath = os.path.join(path, fname)
            if Str in fpath:
                #if test_str not in fpath:
                if meta_str not in fpath:
                    with open(fpath) as f:
                        for l in json_lines.reader(f):
                            #if l["id"] not in id_list:
                            id_list.append(l["id"])
                            questions = l["questions"]
                            context = l["context"].replace("\n", " ").replace(".", "")
                            context = context.replace(",", "")
                            context = context.replace("!", "")
                            context = re.sub(r'\s+', ' ', context)
                            CONTEXT_TEXTS.append(context)
                            # for i in range(len(questions)):
                            #     text = questions[i]["text"]
                            #     label = questions[i]["label"]
                            #     if label == True:
                            #         t = 1
                            #     else:
                            #         t = 0
                            #     q = re.sub(r'\s+', ' ', text)
                            #     q = q.replace(',', '')
                            #     texts.append(context)
                            #     question_list.append(q)
                            #     label_list.append(int(t))
                    f.close()
        # labels.append(label_id)

print('Found %s texts.' % len(CONTEXT_TEXTS))

MAX_NB_WORDS = 20000
MAX_SEQUENCE_LENGTH = 1000
tokenizer = Tokenizer(nb_words=MAX_NB_WORDS,lower=True,filters="")
tokenizer.fit_on_texts(CONTEXT_TEXTS)
# sequences = tokenizer.texts_to_sequences(texts)

WORD_INDEX = tokenizer.word_index
print('Found %s unique tokens.' % len(WORD_INDEX))

==== word_dict_gen_conceptrule.py ====
import os
import json_lines
import re
from keras.preprocessing.text import Tokenizer

# Contextual embeddeding of symbols
texts = []  # list of text samples
id_list = []
question_list = []
label_list = []
labels_index = {}  # dictionary mapping label name to numeric id
labels = []  # list of label ids
#TEXT_DATA_DIR = os.path.abspath('.') + "/data/pararule"
#TEXT_DATA_DIR = "/data/qbao775/deeplogic-master/data/pararule"
TEXT_DATA_DIR = "./data/ConceptRules"

print(TEXT_DATA_DIR)
# TEXT_DATA_DIR = "D:\\AllenAI\\20_newsgroup"
Str = '.jsonl'
CONTEXT_TEXTS = []
#test_str = 'test'
meta_str = 'meta'

for name in sorted(os.listdir(TEXT_DATA_DIR)):
    path = os.path.join(TEXT_DATA_DIR, name)
    if os.path.isdir(path):
        label_id = len(labels_index)
        labels_index[name] = label_id
        for fname in sorted(os.listdir(path)):
            fpath = os.path.join(path, fname)
            if Str in fpath:
                #if test_str not in fpath:
                if meta_str not in fpath:
                    with open(fpath) as f:
                        for l in json_lines.reader(f):
                            #if l["id"] not in id_list:
                            id_list.append(l["id"])
                            questions = l["questions"]
                            context = l["context"].replace("\n", " ").replace(".", "")
                            context = context.replace(",", "")
                            context = context.replace("!", "")
                            context = context.replace("\\", "")
                            context = re.sub(r'\s+', ' ', context)
                            CONTEXT_TEXTS.append(context)
                            # for i in range(len(questions)):
                            #     text = questions[i]["text"]
                            #     label = questions[i]["label"]
                            #     if label == True:
                            #         t = 1
                            #     else:
                            #         t = 0
                            #     q = re.sub(r'\s+', ' ', text)
                            #     q = q.replace(',', '')
                            #     texts.append(context)
                            #     question_list.append(q)
                            #     label_list.append(int(t))
                    f.close()
        # labels.append(label_id)

print('Found %s texts.' % len(CONTEXT_TEXTS))

MAX_NB_WORDS = 20000
MAX_SEQUENCE_LENGTH = 1000
tokenizer = Tokenizer(nb_words=MAX_NB_WORDS,lower=True,filters="")
tokenizer.fit_on_texts(CONTEXT_TEXTS)
# sequences = tokenizer.texts_to_sequences(texts)

WORD_INDEX = tokenizer.word_index
#WORD_INDEX['v'] = 10344
print('Found %s unique tokens.' % len(WORD_INDEX))
#print('Found %s unique tokens.' % len(WORD_INDEX))

==== writer.py ====
#!/usr/bin/env python3
import sys
from pathlib import Path

import numpy as np

# Necessary to load the local gguf package
sys.path.insert(0, str(Path(__file__).parent.parent))

from gguf import GGUFWriter  # noqa: E402


# Example usage:
def writer_example() -> None:
    # Example usage with a file
    gguf_writer = GGUFWriter("example.gguf", "llama")

    gguf_writer.add_block_count(12)
    gguf_writer.add_uint32("answer", 42)  # Write a 32-bit integer
    gguf_writer.add_float32("answer_in_float", 42.0)  # Write a 32-bit float
    gguf_writer.add_custom_alignment(64)
    tensor1 = np.ones((32,), dtype=np.float32) * 100.0
    tensor2 = np.ones((64,), dtype=np.float32) * 101.0
    tensor3 = np.ones((96,), dtype=np.float32) * 102.0

    gguf_writer.add_tensor("tensor1", tensor1)
    gguf_writer.add_tensor("tensor2", tensor2)
    gguf_writer.add_tensor("tensor3", tensor3)

    gguf_writer.write_header_to_file()
    gguf_writer.write_kv_data_to_file()
    gguf_writer.write_tensors_to_file()

    gguf_writer.close()


if __name__ == '__main__':
    writer_example()

==== zerogru.py ====
"""ZeroGRU module for nested RNNs."""
import keras.backend as K
import keras.layers as L

class ZeroGRUCell(L.GRUCell):
  """GRU Cell that skips timestep if inputs are all zero."""
  def call(self, inputs, states, training=None):
    """Step function of the cell."""
    h_tm1 = states[0] # previous output
    cond = K.all(K.equal(inputs, 0), axis=-1)
    new_output, new_states = super().call(inputs, states, training=training)
    curr_output = K.switch(cond, h_tm1, new_output)
    curr_states = [K.switch(cond, states[i], new_states[i]) for i in range(len(states))]
    return curr_output, curr_states



class ZeroGRU(L.GRU):
  """Layer wrapper for the ZeroGRUCell."""
  def __init__(self, units,
               activation='tanh',
               recurrent_activation='hard_sigmoid',
               use_bias=True,
               kernel_initializer='glorot_uniform',
               recurrent_initializer='orthogonal',
               bias_initializer='zeros',
               kernel_regularizer=None,
               recurrent_regularizer=None,
               bias_regularizer=None,
               activity_regularizer=None,
               kernel_constraint=None,
               recurrent_constraint=None,
               bias_constraint=None,
               dropout=0.,
               recurrent_dropout=0.,
               implementation=1,
               return_sequences=False,
               return_state=False,
               go_backwards=False,
               stateful=False,
               unroll=False,
               reset_after=False,
               **kwargs):
    cell = ZeroGRUCell(units,
                       activation=activation,
                       recurrent_activation=recurrent_activation,
                       use_bias=use_bias,
                       kernel_initializer=kernel_initializer,
                       recurrent_initializer=recurrent_initializer,
                       bias_initializer=bias_initializer,
                       kernel_regularizer=kernel_regularizer,
                       recurrent_regularizer=recurrent_regularizer,
                       bias_regularizer=bias_regularizer,
                       kernel_constraint=kernel_constraint,
                       recurrent_constraint=recurrent_constraint,
                       bias_constraint=bias_constraint,
                       dropout=dropout,
                       recurrent_dropout=recurrent_dropout,
                       implementation=implementation,
                       reset_after=reset_after)
    super(L.GRU, self).__init__(cell,
                                return_sequences=return_sequences,
                                return_state=return_state,
                                go_backwards=go_backwards,
                                stateful=stateful,
                                unroll=unroll,
                                **kwargs)
    self.activity_regularizer = L.regularizers.get(activity_regularizer)

class NestedTimeDist(L.TimeDistributed):
  """Nested TimeDistributed wrapper for higher rank tensors."""
  def call(self, inputs, mask=None, training=None, initial_state=None):
    def step(x, _):
      output = self.layer.call(x, mask=mask,
                               training=training,
                               initial_state=initial_state)
      return output, []
    _, outputs, _ = K.rnn(step, inputs,
                          initial_states=[],
                          unroll=False)
    return outputs

  def compute_mask(self, inputs, mask=None):
    return None

==== zip_project.py ====
import zipfile
from pathlib import Path
```

```
def zip_project():
    BASE = Path(".")
    zip_path = BASE.with_suffix(".zip")
    print(f"\n[OK] Zipping project to: {zip_path}")
    with zipfile.ZipFile(zip_path, "w", zipfile.ZIP_DEFLATED) as zipf:
        for file in BASE.rglob("*"):
            if file.is_file():
                zipf.write(file, arcname=file.relative_to(BASE))
    print("[OK] ZIP complete.")

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

~~~~
==== Chat_History_Chunk_10.txt ====
# ? NeuroStore: Seed & Walker Core Logic

## 1. `symbol_seed_generator.py`
```python
import os
import yaml
import hashlib
from datetime import datetime

USE_LLM = False
MODEL_PATH = "models/mistral-7b-q4.gguf"
SEED_OUTPUT_DIR = "fragments/core/"
SEED_COUNT = 100

BASE_SEEDS = [
    "truth is important",
    "conflict creates learning",
    "change is constant",
    "observation precedes action",
    "emotion influences memory",
    "self seeks meaning",
    "logic guides belief",
    "doubt triggers inquiry",
    "energy becomes form",
    "ideas replicate",
    "something must stay_still so everything else can move"
]

def generate_id(content):
    return hashlib.sha256(content.encode()).hexdigest()[:12]

def to_fragment(statement):
    parts = statement.split()
    if len(parts) < 3:
        return None
    subj = parts[0]
    pred = parts[1]
    obj = "_".join(parts[2:])
    return {
        "id": generate_id(statement),
        "predicate": pred,
        "arguments": [subj, obj],
        "confidence": 1.0,
        "emotion": {
            "curiosity": 0.8,
            "certainty": 1.0
        },
        "tags": ["seed", "immutable", "core"],
        "immutable": True,
        "claim": statement,
        "timestamp": datetime.utcnow().isoformat()
    }
~~~~

```python
def save_fragment(fragment, output_dir):
    fname = f"frag_{fragment['id']}.yaml"
    path = os.path.join(output_dir, fname)
    with open(path, 'w') as f:
        yaml.dump(fragment, f)

def generate_symbolic_seeds():
    if not os.path.exists(SEED_OUTPUT_DIR):
        os.makedirs(SEED_OUTPUT_DIR)
    seed_statements = BASE_SEEDS[:SEED_COUNT]
    count = 0
    for stmt in seed_statements:
        frag = to_fragment(stmt)
        if frag:
            save_fragment(frag, SEED_OUTPUT_DIR)
            count += 1
    print(f"Generated {count} symbolic seed fragments in {SEED_OUTPUT_DIR}")

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

## 2. `token_agent.py`

```python
import os
import yaml
import time
import random
from pathlib import Path
from core.cortex_bus import send_message

FRAG_DIR = Path("fragments/core")

class TokenAgent:
    def __init__(self, agent_id="token_agent_01"):
        self.agent_id = agent_id
        self.frag_path = FRAG_DIR
        self.fragment_cache = []

    def load_fragments(self):
        files = list(self.frag_path.glob("*.yaml"))
        random.shuffle(files)
        for f in files:
            with open(f, 'r', encoding='utf-8') as file:
                try:
                    frag = yaml.safe_load(file)
                    if frag:
                        self.fragment_cache.append((f, frag))
                except yaml.YAMLError as e:
                    print(f"[{self.agent_id}] YAML error in {f.name}: {e}")
    def walk_fragment(self, path, frag):
        if 'claim' not in frag:
            return
        walk_log = {
            'fragment': path.name,
            'claim': frag['claim'],
            'tags': frag.get('tags', []),
            'confidence': frag.get('confidence', 0.5),
            'walk_time': time.time()
        }
        if random.random() < 0.2:
            walk_log['flag_mutation'] = True
        send_message({
            'from': self.agent_id,
            'type': 'walk_log',
            'payload': walk_log,
            'timestamp': int(time.time())
        })

    def run(self):
        self.load_fragments()
        for path, frag in self.fragment_cache:
            self.walk_fragment(path, frag)
            time.sleep(0.1)

if __name__ == "__main__":
    agent = TokenAgent()
    agent.run()
```

## 3. `backup_and_export.py`
```python
import os
import tarfile
from datetime import datetime

EXPORT_DIR = os.path.expanduser("~/neurostore/backups")
SOURCE_DIRS = [
    "agents",
    "fragments",
    "logs",
    "meta",
    "runtime",
    "data"
]

os.makedirs(EXPORT_DIR, exist_ok=True)

backup_name = f"neurostore_brain_{datetime.now().strftime('%Y%m%d_%H%M%S')}.tar.gz"
backup_path = os.path.join(EXPORT_DIR, backup_name)

with tarfile.open(backup_path, "w:gz") as tar:
```

~~~~
    for folder in SOURCE_DIRS:
        if os.path.exists(folder):
            print(f"[+] Archiving {folder}/")
            tar.add(folder, arcname=folder)
        else:
            print(f"[-] Skipped missing folder: {folder}")

print(f"
[?] Brain backup complete ? {backup_path}")
```

---

## 4. `deep_file_crawler.py`
```python
import os
import hashlib
from datetime import datetime

def hash_file(path, chunk_size=8192):
    try:
        hasher = hashlib.md5()
        with open(path, 'rb') as f:
            for chunk in iter(lambda: f.read(chunk_size), b""):
                hasher.update(chunk)
        return hasher.hexdigest()
    except Exception as e:
        return f"ERROR: {e}"

def crawl_directory(root_path, out_path):
    count = 0
    with open(out_path, 'w') as out_file:
        for dirpath, dirnames, filenames in os.walk(root_path):
            for file in filenames:
                full_path = os.path.join(dirpath, file)
                try:
                    stat = os.stat(full_path)
                    hashed = hash_file(full_path)
                    line = f"{full_path} | {stat.st_size} bytes | hash: {hashed}"
                except Exception as e:
                    line = f"{full_path} | ERROR: {str(e)}"
                out_file.write(line + "
")
                count += 1
                if count % 100 == 0:
                    print(f"[+] {count} files crawled...")

    print(f"
[?] Crawl complete. Total files: {count}")
    print(f"[?] Full output saved to: {out_path}")

if __name__ == "__main__":
    BASE = "/home/neuroadmin/neurostore"
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
~~~~

```
    output_txt = f"/home/neuroadmin/neurostore_crawl_output_{timestamp}.txt"

    print(f"[*] Starting deep crawl on: {BASE}")
    crawl_directory(BASE, output_txt)
```

## 5. `boot_wrapper.py`
```python
import subprocess
import os
import platform
import time
import psutil
from pathlib import Path

SCRIPTS = [
    "deep_system_scan.py",
    "auto_configurator.py",
    "path_optimizer.py",
    "fragment_teleporter.py",
    "run_logicshredder.py"
]

LOG_PATH = Path("logs/boot_times.log")
LOG_PATH.parent.mkdir(exist_ok=True)

def run_script(name, timings):
    if not Path(name).exists():
        print(f"[boot] ? Missing script: {name}")
        timings.append((name, "MISSING", "-", "-"))
        return False

    print(f"[boot] ? Running: {name}")
    start = time.time()
    proc = psutil.Popen(["python", name])

    peak_mem = 0
    cpu_percent = []

    try:
        while proc.is_running():
            mem = proc.memory_info().rss / (1024**2)
            peak_mem = max(peak_mem, mem)
            cpu = proc.cpu_percent(interval=0.1)
            cpu_percent.append(cpu)
    except Exception:
        pass

    end = time.time()
    duration = round(end - start, 2)
    avg_cpu = round(sum(cpu_percent) / len(cpu_percent), 1) if cpu_percent else 0
```

~~~
    print(f"[boot] ? {name} finished in {duration}s | CPU: {avg_cpu}% | MEM: {int(peak_mem)}MB
")
    timings.append((name, duration, avg_cpu, int(peak_mem)))
    return proc.returncode == 0

def log_timings(timings, total):
    with open(LOG_PATH, "a", encoding="utf-8") as log:
        log.write(f"
=== BOOT TELEMETRY [{time.strftime('%Y-%m-%d %H:%M:%S')}] ===
")
        for name, dur, cpu, mem in timings:
            log.write(f" - {name}: {dur}s | CPU: {cpu}% | MEM: {mem}MB
")
        log.write(f"TOTAL BOOT TIME: {round(total, 2)} seconds
")

def main():
    print("? LOGICSHREDDER SYSTEM BOOT STARTED")
    print(f"? Platform: {platform.system()} | Python: {platform.python_version()}")
    print("==============================================
")

    start_total = time.time()
    timings = []

    for script in SCRIPTS:
        success = run_script(script, timings)
        if not success:
            print(f"[boot] ? Boot aborted due to failure in {script}")
            break

    total_time = time.time() - start_total
    print(f"? BOOT COMPLETE in {round(total_time, 2)} seconds.")
    log_timings(timings, total_time)

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

---

## 6. `quant_feeder_setup.py`
```python
import subprocess
import os
from pathlib import Path
import sys
import time
import urllib.request
import zipfile

LLAMA_REPO = "https://github.com/ggerganov/llama.cpp.git"
MODEL_URL                                                                                       =
"https://huggingface.co/afrideva/Tinystories-gpt-0.1-3m-GGUF/resolve/main/TinyStories-GPT-0.1-3M.Q2_K.gguf"
~~~

```python
MODEL_DIR = Path("models")
MODEL_FILE = MODEL_DIR / "TinyStories.Q2_K.gguf"
LLAMA_DIR = Path("llama.cpp")
LLAMA_BIN = LLAMA_DIR / "build/bin/main"

def install_dependencies():
    print("[setup] ? Installing dependencies...")
    subprocess.run([sys.executable, "-m", "pip", "install", "--quiet", "--upgrade", "pip"])
    subprocess.run([sys.executable, "-m", "pip", "install", "--quiet", "requests"])

def clone_llama_cpp():
    if not LLAMA_DIR.exists():
        print("[setup] ? Cloning llama.cpp...")
        subprocess.run(["git", "clone", LLAMA_REPO])
    else:
        print("[setup] ? llama.cpp already exists")

def build_llama_cpp():
    print("[setup] ? Building llama.cpp...")
    os.makedirs(LLAMA_DIR / "build", exist_ok=True)
    subprocess.run(["cmake", "-B", "build"], cwd=LLAMA_DIR)
    subprocess.run(["cmake", "--build", "build", "--config", "Release"], cwd=LLAMA_DIR)

def download_model():
    if MODEL_FILE.exists():
        print(f"[setup] ? Model already downloaded: {MODEL_FILE.name}")
        return
    print(f"[setup] ??  Downloading model to {MODEL_FILE}...")
    MODEL_DIR.mkdir(parents=True, exist_ok=True)
    urllib.request.urlretrieve(MODEL_URL, MODEL_FILE)

def patch_feeder():
    print("[setup] ?? Patching quant_prompt_feeder.py with model and llama path")
    feeder_code = Path("quant_prompt_feeder.py").read_text(encoding="utf-8")
    patched = feeder_code.replace(
        'MODEL_PATH = Path("models/TinyLlama.Q4_0.gguf")',
        f'MODEL_PATH = Path("{MODEL_FILE.as_posix()}")'
    ).replace(
        'LLAMA_CPP_PATH = Path("llama.cpp/build/bin/main")',
        f'LLAMA_CPP_PATH = Path("{LLAMA_BIN.as_posix()}")'
    )
    Path("quant_prompt_feeder.py").write_text(patched, encoding="utf-8")

def run_feeder():
    print("[setup] ? Running quant_prompt_feeder.py...")
    subprocess.run(["python", "quant_prompt_feeder.py"])

if __name__ == "__main__":
    install_dependencies()
    clone_llama_cpp()
    build_llama_cpp()
    download_model()
    patch_feeder()
    run_feeder()
```

## 7. `benchmark_agent.py`
```python
import time
import random
import psutil
import threading

results = {}

def simulate_fragment_walks(num_fragments, walk_speed_per_sec):
    walks_done = 0
    start_time = time.time()
    end_time = start_time + 10
    while time.time() < end_time:
        walks_done += walk_speed_per_sec
        time.sleep(1)
    results['walks'] = walks_done

def simulate_mutation_ops(rate_per_sec):
    mutations_done = 0
    start_time = time.time()
    end_time = start_time + 10
    while time.time() < end_time:
        mutations_done += rate_per_sec
        time.sleep(1)
    results['mutations'] = mutations_done

def simulate_emotion_decay_ops(fragments_count, decay_passes_per_sec):
    decay_ops_done = 0
    start_time = time.time()
    end_time = start_time + 10
    while time.time() < end_time:
        decay_ops_done += decay_passes_per_sec
        time.sleep(1)
    results['decay'] = decay_ops_done

def run():
    walk_thread = threading.Thread(target=simulate_fragment_walks, args=(10000, random.randint(200, 350)))
    mutate_thread = threading.Thread(target=simulate_mutation_ops, args=(random.randint(30, 60),))
    decay_thread = threading.Thread(target=simulate_emotion_decay_ops, args=(10000, random.randint(50, 100)))

    walk_thread.start()
    mutate_thread.start()
    decay_thread.start()

    walk_thread.join()
    mutate_thread.join()
    decay_thread.join()
    results['cpu_usage_percent'] = psutil.cpu_percent(interval=1)
    results['ram_usage_percent'] = psutil.virtual_memory().percent

    print("===== Symbolic TPS Benchmark =====")
    print(f"Fragment Walks     : {results['walks'] // 10} per second")
    print(f"Mutations          : {results['mutations'] // 10} per second")
    print(f"Emotion Decay Ops  : {results['decay'] // 10} per second")
    print()
    print(f"CPU Usage          : {results['cpu_usage_percent']}%")
    print(f"RAM Usage          : {results['ram_usage_percent']}%")
    print("==================================")

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

## 8. `nvme_memory_shim.py`
```python
import os
import time
import yaml
import psutil
from pathlib import Path
from shutil import disk_usage

BASE = Path(__file__).parent
CONFIG_PATH = BASE / "system_config.yaml"
LOGIC_CACHE = BASE / "hotcache"

# ? Improved detection with fallback by mount label
def detect_nvmes():
    nvmes = []
    fallback_mounts = ['C', 'D', 'E', 'F']

    for part in psutil.disk_partitions():
        label = part.device.lower()
        try:
            usage = disk_usage(part.mountpoint)
            is_nvme = any(x in label for x in ['nvme', 'ssd'])
            is_fallback = part.mountpoint.strip(':\').upper() in fallback_mounts

            if is_nvme or is_fallback:
                nvmes.append({
                    'mount': part.mountpoint,
                    'fstype': part.fstype,
                    'free_gb': round(usage.free / 1e9, 2),
                    'total_gb': round(usage.total / 1e9, 2)
                })
        except Exception:
            continue

    print(f"[shim] Detected {len(nvmes)} logic-capable drive(s): {[n['mount'] for n in nvmes]}")
    return sorted(nvmes, key=lambda d: d['free_gb'], reverse=True)

def assign_as_logic_ram(nvmes):
    logic_zones = {}
    for i, nvme in enumerate(nvmes[:4]):  # limit to 4 shards
        zone = f"ram_shard_{i+1}"
        path = Path(nvme['mount']) / "logicshred_cache"
        path.mkdir(exist_ok=True)
        logic_zones[zone] = str(path)
    return logic_zones

def update_config(zones):
    if CONFIG_PATH.exists():
        with open(CONFIG_PATH, 'r') as f:
            config = yaml.safe_load(f)
    else:
        config = {}

    config['logic_ram'] = zones
    config['hotcache_path'] = str(LOGIC_CACHE)
    with open(CONFIG_PATH, 'w') as f:
        yaml.safe_dump(config, f)
    print(f"? Config updated with NVMe logic cache: {list(zones.values())}")

if __name__ == "__main__":
    LOGIC_CACHE.mkdir(exist_ok=True)
    print("? Detecting NVMe drives and logic RAM mounts...")
    drives = detect_nvmes()
    if not drives:
        print("?? No NVMe or fallback drives detected. System unchanged.")
    else:
        zones = assign_as_logic_ram(drives)
        update_config(zones)
```

## 9. `layer_inference_engine.py`

```python
import os
import numpy as np
from concurrent.futures import ThreadPoolExecutor
from collections import OrderedDict

# ========== I/O FUNCTIONS ==========

def load_embedding(token_id, path="/NeuroStore/embeddings"):
    filepath = os.path.join(path, f"{token_id}.bin")
    return np.fromfile(filepath, dtype=np.float32)

def load_layer_weights(layer_id, base="/NeuroStore/layers"):
    layer_dir = os.path.join(base, f"layer_{layer_id:04d}")
    attention = np.fromfile(os.path.join(layer_dir, "attention_weights.bin"), dtype=np.float32)
    feedforward = np.fromfile(os.path.join(layer_dir, "feedforward_weights.bin"), dtype=np.float32)
    return attention.reshape(768, 768), feedforward.reshape(768, 768)

# ========== COMPUTATION ==========

def forward_pass(embedding, layer_weights):
    attention, feedforward = layer_weights
    attention_result = np.dot(embedding, attention)
    return np.dot(attention_result, feedforward)

def load_layers_in_parallel(layer_ids):
    with ThreadPoolExecutor() as executor:
        return list(executor.map(load_layer_weights, layer_ids))

# ========== MEMORY ==========

class LRUCache(OrderedDict):
    def __init__(self, capacity):
        super().__init__()
        self.capacity = capacity

    def get(self, key):
        if key in self:
            self.move_to_end(key)
            return self[key]
        return None

    def put(self, key, value):
        if len(self) >= self.capacity:
            self.popitem(last=False)
        self[key] = value

# ========== SAMPLE INIT ==========

def generate_sample_files():
    os.makedirs("/NeuroStore/embeddings", exist_ok=True)
    os.makedirs("/NeuroStore/layers/layer_0001", exist_ok=True)

    embedding = np.random.rand(768).astype(np.float32)
    embedding.tofile("/NeuroStore/embeddings/token_001.bin")

    attn = np.random.rand(768, 768).astype(np.float32)
    ffwd = np.random.rand(768, 768).astype(np.float32)

    attn.tofile("/NeuroStore/layers/layer_0001/attention_weights.bin")
    ffwd.tofile("/NeuroStore/layers/layer_0001/feedforward_weights.bin")

# ========== USAGE EXAMPLE ==========

if __name__ == "__main__":
    generate_sample_files()
    embedding = load_embedding("token_001")
    layer_weights = load_layer_weights(1)
    output = forward_pass(embedding, layer_weights)
    print("Forward pass output shape:", output.shape)
```

## 10. `memory_tracker.py`
```python
import psutil
import time
from datetime import datetime
from pathlib import Path

LOG_PATH = Path("logs/memory_usage.log")
LOG_PATH.parent.mkdir(exist_ok=True)

class MemoryTracker:
    def __init__(self, interval=5):
        self.interval = interval

    def log_memory(self):
        while True:
            usage = psutil.virtual_memory()
             log_line = f"[{datetime.now().isoformat()}] RAM: {usage.percent}% used of {usage.total / 1e9:.2f}
GB
"
            with open(LOG_PATH, 'a') as log:
                log.write(log_line)
            print(log_line.strip())
            time.sleep(self.interval)

if __name__ == "__main__":
    tracker = MemoryTracker(interval=10)
    tracker.log_memory()
```

## 11. `memory_archiver.py`
```python
import os
import shutil
import time
from datetime import datetime
from pathlib import Path

SOURCE_DIR = Path("hotcache")
ARCHIVE_ROOT = Path("archive/memory")
ARCHIVE_ROOT.mkdir(parents=True, exist_ok=True)

INTERVAL_SECONDS = 60 * 15  # every 15 minutes

print("[ARCHIVER] Starting memory snapshot loop...")
while True:
    if not SOURCE_DIR.exists():
        print("[ARCHIVER] Source cache not found. Waiting...")
        time.sleep(INTERVAL_SECONDS)
        continue

    stamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    dest = ARCHIVE_ROOT / f"snapshot_{stamp}"
    shutil.copytree(SOURCE_DIR, dest)
    print(f"[ARCHIVER] Snapshot saved ? {dest}")
    time.sleep(INTERVAL_SECONDS)
```

## 12. `memory_visualizer.py`
```python
import os
import yaml
import matplotlib.pyplot as plt
from pathlib import Path

FRAG_PATH = Path("fragments/core")

# Count frequency of each tag
tag_freq = {}
conf_values = []

for file in FRAG_PATH.glob("*.yaml"):
    try:
        with open(file, 'r') as f:
            frag = yaml.safe_load(f)
            tags = frag.get("tags", [])
            conf = frag.get("confidence", 0.5)
            conf_values.append(conf)
            for tag in tags:
                tag_freq[tag] = tag_freq.get(tag, 0) + 1
    except Exception as e:
        print(f"Error reading {file}: {e}")

# Plot tag distribution
plt.figure(figsize=(10, 4))
plt.bar(tag_freq.keys(), tag_freq.values(), color='skyblue')
plt.xticks(rotation=45)
plt.title("Tag Frequency in Symbolic Fragments")
plt.tight_layout()
plt.savefig("logs/tag_frequency_plot.png")
plt.close()

# Plot confidence histogram
plt.figure(figsize=(6, 4))
plt.hist(conf_values, bins=20, color='salmon', edgecolor='black')
plt.title("Confidence Score Distribution")
plt.xlabel("Confidence")
plt.ylabel("Count")
plt.tight_layout()
plt.savefig("logs/confidence_histogram.png")
plt.close()

print("[Visualizer] Tag frequency and confidence distribution plots saved to logs/.")
```
```

==== compile_to_pdf.py ====
import os
from pathlib import Path
from fpdf import FPDF

Extensions to include
FILE_EXTENSIONS = [".py", ".yaml", ".yml", ".json", ".txt"]

class CodePDF(FPDF):
    def __init__(self):
        super().__init__()
        self.set_auto_page_break(auto=True, margin=15)
        self.add_page()
        self.set_font("Courier", size=8)

    def add_code_file(self, filepath):
        self.set_font("Courier", size=8)
        self.multi_cell(0, 5, f"\n==== {filepath} ====\n")
        try:
            with open(filepath, 'r', encoding='utf-8', errors='ignore') as f:
                for line in f:
                    clean_line = ''.join(c if 0x20 <= ord(c) <= 0x7E or c in '\t\n\r' else '?' for c in line)
                    self.multi_cell(0, 5, clean_line.rstrip())
        except Exception as e:
            self.multi_cell(0, 5, f"[Error reading {filepath}: {e}]\n")

def gather_files(root_dir, extensions):
    return [
        f for f in Path(root_dir).rglob("*")
         if f.is_file() and f.suffix.lower() in extensions and "venv" not in f.parts and "__pycache__" not in
f.parts
    ]

def main(root=".", output="symbolic_manifesto.pdf"):
    pdf = CodePDF()
    files = gather_files(root, FILE_EXTENSIONS)

    if not files:
        print("[!] No matching files found.")
        return

    for file in sorted(files):
        pdf.add_code_file(file)

    pdf.output(output)
    print(f"[?] Compiled {len(files)} files into: {output}")

if __name__ == "__main__":
    main()

==== FULL_MANIFEST.txt ====
import os
import yaml
import hashlib
from datetime import datetime

USE_LLM = False
MODEL_PATH = "models/mistral-7b-q4.gguf"
SEED_OUTPUT_DIR = "fragments/core/"
SEED_COUNT = 100

BASE_SEEDS = [
    "truth is important",
    "conflict creates learning",
    "change is constant",
    "observation precedes action",
    "emotion influences memory",
    "self seeks meaning",
    "logic guides belief",
    "doubt triggers inquiry",
    "energy becomes form",
    "ideas replicate",
    "something must stay_still so everything else can move"
]

def generate_id(content):
    return hashlib.sha256(content.encode()).hexdigest()[:12]

def to_fragment(statement):
    parts = statement.split()
    if len(parts) < 3:
        return None
    subj = parts[0]
    pred = parts[1]
    obj = "_".join(parts[2:])
    return {
        "id": generate_id(statement),
        "predicate": pred,
        "arguments": [subj, obj],
        "confidence": 1.0,
        "emotion": {
            "curiosity": 0.8,
            "certainty": 1.0
        },
        "tags": ["seed", "immutable", "core"],
        "immutable": True,
        "claim": statement,
        "timestamp": datetime.utcnow().isoformat()
    }

def save_fragment(fragment, output_dir):
    fname = f"frag_{fragment['id']}.yaml"
    path = os.path.join(output_dir, fname)
    with open(path, 'w') as f:
        yaml.dump(fragment, f)

def generate_symbolic_seeds():
    if not os.path.exists(SEED_OUTPUT_DIR):
        os.makedirs(SEED_OUTPUT_DIR)
    seed_statements = BASE_SEEDS[:SEED_COUNT]
    count = 0
    for stmt in seed_statements:
        frag = to_fragment(stmt)
        if frag:
            save_fragment(frag, SEED_OUTPUT_DIR)
            count += 1
    print(f"Generated {count} symbolic seed fragments in {SEED_OUTPUT_DIR}")

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

```
 import time
import random
from pathlib import Path
from core.utils import load_yaml, validate_fragment
from core.cortex_bus import send_message

FRAG_DIR = Path("fragments/core")

class TokenAgent:
    def __init__(self, agent_id="token_agent_01"):
        self.agent_id = agent_id
        self.frag_path = FRAG_DIR
        self.fragment_cache = []

    def load_fragments(self):
        files = list(self.frag_path.glob("*.yaml"))
        random.shuffle(files)
        for f in files:
            frag = load_yaml(f, validate_schema=validate_fragment)
            if frag:
                self.fragment_cache.append((f, frag))

    def walk_fragment(self, path, frag):
        if 'claim' not in frag:
            return
        walk_log = {
            'fragment': path.name,
            'claim': frag['claim'],
            'tags': frag.get('tags', []),
            'confidence': frag.get('confidence', 0.5),
            'walk_time': time.time()
        }
        if random.random() < 0.2:
```

~~~
            walk_log['flag_mutation'] = True
        send_message({
            'from': self.agent_id,
            'type': 'walk_log',
            'payload': walk_log,
            'timestamp': int(time.time())
        })

    def run(self):
        self.load_fragments()
        for path, frag in self.fragment_cache:
            self.walk_fragment(path, frag)
            time.sleep(0.1)

if __name__ == "__main__":
    agent = TokenAgent()
    agent.run()
```python
import os
import yaml
import time
import random
from pathlib import Path
from core.cortex_bus import send_message

FRAG_DIR = Path("fragments/core")

class TokenAgent:
    def __init__(self, agent_id="token_agent_01"):
        self.agent_id = agent_id
        self.frag_path = FRAG_DIR
        self.fragment_cache = []

    def load_fragments(self):
        files = list(self.frag_path.glob("*.yaml"))
        random.shuffle(files)
        for f in files:
            with open(f, 'r', encoding='utf-8') as file:
                try:
                    frag = yaml.safe_load(file)
                    if frag:
                        self.fragment_cache.append((f, frag))
                except yaml.YAMLError as e:
                    print(f"[{self.agent_id}] YAML error in {f.name}: {e}")

    def walk_fragment(self, path, frag):
        if 'claim' not in frag:
            return
        walk_log = {
            'fragment': path.name,
            'claim': frag['claim'],
            'tags': frag.get('tags', []),
            'confidence': frag.get('confidence', 0.5),
            'walk_time': time.time()
~~~

```
        }
        if random.random() < 0.2:
            walk_log['flag_mutation'] = True
        send_message({
            'from': self.agent_id,
            'type': 'walk_log',
            'payload': walk_log,
            'timestamp': int(time.time())
        })

    def run(self):
        self.load_fragments()
        for path, frag in self.fragment_cache:
            self.walk_fragment(path, frag)
            time.sleep(0.1)

if __name__ == "__main__":
    agent = TokenAgent()
    agent.run()
```

```
 import os
import tarfile
from datetime import datetime

EXPORT_DIR = os.path.expanduser("~/neurostore/backups")
SOURCE_DIRS = [
    "agents",
    "fragments",
    "logs",
    "meta",
    "runtime",
    "data"
]

os.makedirs(EXPORT_DIR, exist_ok=True)

backup_name = f"neurostore_brain_{datetime.now().strftime('%Y%m%d_%H%M%S')}.tar.gz"
backup_path = os.path.join(EXPORT_DIR, backup_name)

with tarfile.open(backup_path, "w:gz") as tar:
    for folder in SOURCE_DIRS:
        if os.path.exists(folder):
            print(f"[+] Archiving {folder}/")
            tar.add(folder, arcname=folder)
        else:
            print(f"[-] Skipped missing folder: {folder}")

print(f"
[?] Brain backup complete ? {backup_path}")
```

```
import os
import hashlib
from datetime import datetime

def hash_file(path, chunk_size=8192):
    try:
        hasher = hashlib.md5()
        with open(path, 'rb') as f:
            for chunk in iter(lambda: f.read(chunk_size), b""):
                hasher.update(chunk)
        return hasher.hexdigest()
    except Exception as e:
        return f"ERROR: {e}"

def crawl_directory(root_path, out_path):
    count = 0
    with open(out_path, 'w') as out_file:
        for dirpath, dirnames, filenames in os.walk(root_path):
            for file in filenames:
                full_path = os.path.join(dirpath, file)
                try:
                    stat = os.stat(full_path)
                    hashed = hash_file(full_path)
                    line = f"{full_path} | {stat.st_size} bytes | hash: {hashed}"
                except Exception as e:
                    line = f"{full_path} | ERROR: {str(e)}"
                out_file.write(line + "
")
                count += 1
                if count % 100 == 0:
                    print(f"[+] {count} files crawled...")

    print(f"
[?] Crawl complete. Total files: {count}")
    print(f"[?] Full output saved to: {out_path}")

if __name__ == "__main__":
    BASE = "/home/neuroadmin/neurostore"
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    output_txt = f"/home/neuroadmin/neurostore_crawl_output_{timestamp}.txt"

    print(f"[*] Starting deep crawl on: {BASE}")
    crawl_directory(BASE, output_txt)
```

```
 import subprocess
import os
import platform
import time
import psutil
from pathlib import Path

SCRIPTS = [
    "deep_system_scan.py",
    "auto_configurator.py",
    "path_optimizer.py",
    "fragment_teleporter.py",
    "run_logicshredder.py"
]

LOG_PATH = Path("logs/boot_times.log")
LOG_PATH.parent.mkdir(exist_ok=True)

def run_script(name, timings):
    if not Path(name).exists():
        print(f"[boot] ? Missing script: {name}")
        timings.append((name, "MISSING", "-", "-"))
        return False

    print(f"[boot] ? Running: {name}")
    start = time.time()
    proc = psutil.Popen(["python", name])

    peak_mem = 0
    cpu_percent = []

    try:
        while proc.is_running():
            mem = proc.memory_info().rss / (1024**2)
            peak_mem = max(peak_mem, mem)
            cpu = proc.cpu_percent(interval=0.1)
            cpu_percent.append(cpu)
    except Exception:
        pass

    end = time.time()
    duration = round(end - start, 2)
    avg_cpu = round(sum(cpu_percent) / len(cpu_percent), 1) if cpu_percent else 0

    print(f"[boot] ? {name} finished in {duration}s | CPU: {avg_cpu}% | MEM: {int(peak_mem)}MB
")
    timings.append((name, duration, avg_cpu, int(peak_mem)))
    return proc.returncode == 0

def log_timings(timings, total):
    with open(LOG_PATH, "a", encoding="utf-8") as log:
        log.write(f"
=== BOOT TELEMETRY [{time.strftime('%Y-%m-%d %H:%M:%S')}] ===
")
        for name, dur, cpu, mem in timings:
            log.write(f" - {name}: {dur}s | CPU: {cpu}% | MEM: {mem}MB
")
        log.write(f"TOTAL BOOT TIME: {round(total, 2)} seconds
")

def main():
    print("? LOGICSHREDDER SYSTEM BOOT STARTED")
    print(f"? Platform: {platform.system()} | Python: {platform.python_version()}")
    print("=============================================
")

    start_total = time.time()
    timings = []

    for script in SCRIPTS:
        success = run_script(script, timings)
        if not success:
            print(f"[boot] ? Boot aborted due to failure in {script}")
            break

    total_time = time.time() - start_total
    print(f"? BOOT COMPLETE in {round(total_time, 2)} seconds.")
    log_timings(timings, total_time)

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

```
 import subprocess
import os
from pathlib import Path
import sys
import time
import urllib.request
import zipfile

LLAMA_REPO = "https://github.com/ggerganov/llama.cpp.git"
MODEL_URL                                                                                                                          =
"https://huggingface.co/afrideva/Tinystories-gpt-0.1-3m-GGUF/resolve/main/TinyStories-GPT-0.1-3M.Q2_K.gguf"

MODEL_DIR = Path("models")
MODEL_FILE = MODEL_DIR / "TinyStories.Q2_K.gguf"
LLAMA_DIR = Path("llama.cpp")
LLAMA_BIN = LLAMA_DIR / "build/bin/main"

def install_dependencies():
    print("[setup] ? Installing dependencies...")
    subprocess.run([sys.executable, "-m", "pip", "install", "--quiet", "--upgrade", "pip"])
    subprocess.run([sys.executable, "-m", "pip", "install", "--quiet", "requests"])

def clone_llama_cpp():
    if not LLAMA_DIR.exists():
        print("[setup] ? Cloning llama.cpp...")
        subprocess.run(["git", "clone", LLAMA_REPO])
    else:
        print("[setup] ? llama.cpp already exists")

def build_llama_cpp():
    print("[setup] ? Building llama.cpp...")
    os.makedirs(LLAMA_DIR / "build", exist_ok=True)
    subprocess.run(["cmake", "-B", "build"], cwd=LLAMA_DIR)
    subprocess.run(["cmake", "--build", "build", "--config", "Release"], cwd=LLAMA_DIR)

def download_model():
    if MODEL_FILE.exists():
        print(f"[setup] ? Model already downloaded: {MODEL_FILE.name}")
        return
    print(f"[setup] ??  Downloading model to {MODEL_FILE}...")
    MODEL_DIR.mkdir(parents=True, exist_ok=True)
    urllib.request.urlretrieve(MODEL_URL, MODEL_FILE)

def patch_feeder():
    print("[setup] ?? Patching quant_prompt_feeder.py with model and llama path")
    feeder_code = Path("quant_prompt_feeder.py").read_text(encoding="utf-8")
    patched = feeder_code.replace(
        'MODEL_PATH = Path("models/TinyLlama.Q4_0.gguf")',
        f'MODEL_PATH = Path("{MODEL_FILE.as_posix()}")'
    ).replace(
        'LLAMA_CPP_PATH = Path("llama.cpp/build/bin/main")',
        f'LLAMA_CPP_PATH = Path("{LLAMA_BIN.as_posix()}")'
    )
    Path("quant_prompt_feeder.py").write_text(patched, encoding="utf-8")

def run_feeder():
    print("[setup] ? Running quant_prompt_feeder.py...")
    subprocess.run(["python", "quant_prompt_feeder.py"])

if __name__ == "__main__":
    install_dependencies()
    clone_llama_cpp()
    build_llama_cpp()
    download_model()
    patch_feeder()
    run_feeder()
```

```python
 import time
import random
import psutil
import threading

results = {}

def simulate_fragment_walks(num_fragments, walk_speed_per_sec):
    walks_done = 0
    start_time = time.time()
    end_time = start_time + 10
    while time.time() < end_time:
        walks_done += walk_speed_per_sec
        time.sleep(1)
    results['walks'] = walks_done

def simulate_mutation_ops(rate_per_sec):
    mutations_done = 0
    start_time = time.time()
    end_time = start_time + 10
    while time.time() < end_time:
        mutations_done += rate_per_sec
        time.sleep(1)
    results['mutations'] = mutations_done

def simulate_emotion_decay_ops(fragments_count, decay_passes_per_sec):
    decay_ops_done = 0
    start_time = time.time()
    end_time = start_time + 10
    while time.time() < end_time:
        decay_ops_done += decay_passes_per_sec
        time.sleep(1)
    results['decay'] = decay_ops_done

def run():
    walk_thread = threading.Thread(target=simulate_fragment_walks, args=(10000, random.randint(200, 350)))
    mutate_thread = threading.Thread(target=simulate_mutation_ops, args=(random.randint(30, 60),))
    decay_thread = threading.Thread(target=simulate_emotion_decay_ops, args=(10000, random.randint(50, 100)))

    walk_thread.start()
    mutate_thread.start()
    decay_thread.start()

    walk_thread.join()
    mutate_thread.join()
    decay_thread.join()

    results['cpu_usage_percent'] = psutil.cpu_percent(interval=1)
    results['ram_usage_percent'] = psutil.virtual_memory().percent

    print("===== Symbolic TPS Benchmark =====")
    print(f"Fragment Walks     : {results['walks'] // 10} per second")
    print(f"Mutations          : {results['mutations'] // 10} per second")
    print(f"Emotion Decay Ops  : {results['decay'] // 10} per second")
    print()
    print(f"CPU Usage          : {results['cpu_usage_percent']}%")
    print(f"RAM Usage          : {results['ram_usage_percent']}%")
    print("=================================")

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

```
 import os
import time
import yaml
import psutil
from pathlib import Path
from shutil import disk_usage

BASE = Path(__file__).parent
CONFIG_PATH = BASE / "system_config.yaml"
LOGIC_CACHE = BASE / "hotcache"

# ? Improved detection with fallback by mount label
def detect_nvmes():
    nvmes = []
    fallback_mounts = ['C', 'D', 'E', 'F']

    for part in psutil.disk_partitions():
        label = part.device.lower()
        try:
            usage = disk_usage(part.mountpoint)
            is_nvme = any(x in label for x in ['nvme', 'ssd'])
            is_fallback = part.mountpoint.strip(':\').upper() in fallback_mounts

            if is_nvme or is_fallback:
                nvmes.append({
                    'mount': part.mountpoint,
                    'fstype': part.fstype,
                    'free_gb': round(usage.free / 1e9, 2),
                    'total_gb': round(usage.total / 1e9, 2)
                })
        except Exception:
            continue

    print(f"[shim] Detected {len(nvmes)} logic-capable drive(s): {[n['mount'] for n in nvmes]}")
    return sorted(nvmes, key=lambda d: d['free_gb'], reverse=True)

def assign_as_logic_ram(nvmes):
    logic_zones = {}
    for i, nvme in enumerate(nvmes[:4]):  # limit to 4 shards
        zone = f"ram_shard_{i+1}"
        path = Path(nvme['mount']) / "logicshred_cache"
        path.mkdir(exist_ok=True)
        logic_zones[zone] = str(path)
    return logic_zones

def update_config(zones):
    if CONFIG_PATH.exists():
        with open(CONFIG_PATH, 'r') as f:
            config = yaml.safe_load(f)
    else:
        config = {}

    config['logic_ram'] = zones
    config['hotcache_path'] = str(LOGIC_CACHE)
    with open(CONFIG_PATH, 'w') as f:
        yaml.safe_dump(config, f)
    print(f"? Config updated with NVMe logic cache: {list(zones.values())}")

if __name__ == "__main__":
    LOGIC_CACHE.mkdir(exist_ok=True)
    print("? Detecting NVMe drives and logic RAM mounts...")
    drives = detect_nvmes()
    if not drives:
        print("?? No NVMe or fallback drives detected. System unchanged.")
    else:
        zones = assign_as_logic_ram(drives)
        update_config(zones)
```

```
  import os
import numpy as np
from concurrent.futures import ThreadPoolExecutor
from collections import OrderedDict

# ========== I/O FUNCTIONS ==========

def load_embedding(token_id, path="/NeuroStore/embeddings"):
    filepath = os.path.join(path, f"{token_id}.bin")
    return np.fromfile(filepath, dtype=np.float32)

def load_layer_weights(layer_id, base="/NeuroStore/layers"):
    layer_dir = os.path.join(base, f"layer_{layer_id:04d}")
    attention = np.fromfile(os.path.join(layer_dir, "attention_weights.bin"), dtype=np.float32)
    feedforward = np.fromfile(os.path.join(layer_dir, "feedforward_weights.bin"), dtype=np.float32)
    return attention.reshape(768, 768), feedforward.reshape(768, 768)

# ========== COMPUTATION ==========

def forward_pass(embedding, layer_weights):
    attention, feedforward = layer_weights
    attention_result = np.dot(embedding, attention)
    return np.dot(attention_result, feedforward)

def load_layers_in_parallel(layer_ids):
    with ThreadPoolExecutor() as executor:
        return list(executor.map(load_layer_weights, layer_ids))

# ========== MEMORY ==========

class LRUCache(OrderedDict):
    def __init__(self, capacity):
        super().__init__()
        self.capacity = capacity

    def get(self, key):
        if key in self:
            self.move_to_end(key)
            return self[key]
        return None

    def put(self, key, value):
        if len(self) >= self.capacity:
            self.popitem(last=False)
        self[key] = value

# ========== SAMPLE INIT ==========

def generate_sample_files():
    os.makedirs("/NeuroStore/embeddings", exist_ok=True)
    os.makedirs("/NeuroStore/layers/layer_0001", exist_ok=True)

    embedding = np.random.rand(768).astype(np.float32)
    embedding.tofile("/NeuroStore/embeddings/token_001.bin")

    attn = np.random.rand(768, 768).astype(np.float32)
    ffwd = np.random.rand(768, 768).astype(np.float32)

    attn.tofile("/NeuroStore/layers/layer_0001/attention_weights.bin")
    ffwd.tofile("/NeuroStore/layers/layer_0001/feedforward_weights.bin")

# ========== USAGE EXAMPLE ==========

if __name__ == "__main__":
    generate_sample_files()
    embedding = load_embedding("token_001")
    layer_weights = load_layer_weights(1)
    output = forward_pass(embedding, layer_weights)
    print("Forward pass output shape:", output.shape)
```

```
 import os
import numpy as np
from concurrent.futures import ThreadPoolExecutor
from collections import OrderedDict

# ========== I/O FUNCTIONS ==========

def load_embedding(token_id, path="/NeuroStore/embeddings"):
    filepath = os.path.join(path, f"{token_id}.bin")
    return np.fromfile(filepath, dtype=np.float32)

def load_layer_weights(layer_id, base="/NeuroStore/layers"):
    layer_dir = os.path.join(base, f"layer_{layer_id:04d}")
    attention = np.fromfile(os.path.join(layer_dir, "attention_weights.bin"), dtype=np.float32)
    feedforward = np.fromfile(os.path.join(layer_dir, "feedforward_weights.bin"), dtype=np.float32)
    return attention.reshape(768, 768), feedforward.reshape(768, 768)

# ========== COMPUTATION ==========

def forward_pass(embedding, layer_weights):
    attention, feedforward = layer_weights
    attention_result = np.dot(embedding, attention)
    return np.dot(attention_result, feedforward)

def load_layers_in_parallel(layer_ids):
    with ThreadPoolExecutor() as executor:
        return list(executor.map(load_layer_weights, layer_ids))

# ========== MEMORY ==========

class LRUCache(OrderedDict):
    def __init__(self, capacity):
        super().__init__()
        self.capacity = capacity

    def get(self, key):
        if key in self:
            self.move_to_end(key)
            return self[key]
        return None

    def put(self, key, value):
        if len(self) >= self.capacity:
            self.popitem(last=False)
        self[key] = value

# ========== SAMPLE INIT ==========

def generate_sample_files():
    os.makedirs("/NeuroStore/embeddings", exist_ok=True)
    os.makedirs("/NeuroStore/layers/layer_0001", exist_ok=True)

    embedding = np.random.rand(768).astype(np.float32)
    embedding.tofile("/NeuroStore/embeddings/token_001.bin")

    attn = np.random.rand(768, 768).astype(np.float32)
    ffwd = np.random.rand(768, 768).astype(np.float32)

    attn.tofile("/NeuroStore/layers/layer_0001/attention_weights.bin")
    ffwd.tofile("/NeuroStore/layers/layer_0001/feedforward_weights.bin")

# ========== USAGE EXAMPLE ==========

if __name__ == "__main__":
    generate_sample_files()
    embedding = load_embedding("token_001")
    layer_weights = load_layer_weights(1)
    output = forward_pass(embedding, layer_weights)
    print("Forward pass output shape:", output.shape)

 import os
import shutil
import time
from datetime import datetime
from pathlib import Path

SOURCE_DIR = Path("hotcache")
ARCHIVE_ROOT = Path("archive/memory")
ARCHIVE_ROOT.mkdir(parents=True, exist_ok=True)

INTERVAL_SECONDS = 60 * 15  # every 15 minutes

print("[ARCHIVER] Starting memory snapshot loop...")
while True:
    if not SOURCE_DIR.exists():
        print("[ARCHIVER] Source cache not found. Waiting...")
        time.sleep(INTERVAL_SECONDS)
        continue

    stamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    dest = ARCHIVE_ROOT / f"snapshot_{stamp}"
    shutil.copytree(SOURCE_DIR, dest)
    print(f"[ARCHIVER] Snapshot saved ? {dest}")
    time.sleep(INTERVAL_SECONDS)

 import os
import yaml
import matplotlib.pyplot as plt
from pathlib import Path

FRAG_PATH = Path("fragments/core")

# Count frequency of each tag
tag_freq = {}
conf_values = []

for file in FRAG_PATH.glob("*.yaml"):
    try:
        with open(file, 'r') as f:
            frag = yaml.safe_load(f)
            tags = frag.get("tags", [])
            conf = frag.get("confidence", 0.5)
            conf_values.append(conf)
            for tag in tags:
                tag_freq[tag] = tag_freq.get(tag, 0) + 1
    except Exception as e:
        print(f"Error reading {file}: {e}")

# Plot tag distribution
plt.figure(figsize=(10, 4))
plt.bar(tag_freq.keys(), tag_freq.values(), color='skyblue')
plt.xticks(rotation=45)
plt.title("Tag Frequency in Symbolic Fragments")
plt.tight_layout()
plt.savefig("logs/tag_frequency_plot.png")
plt.close()

# Plot confidence histogram
plt.figure(figsize=(6, 4))
plt.hist(conf_values, bins=20, color='salmon', edgecolor='black')
plt.title("Confidence Score Distribution")
plt.xlabel("Confidence")
plt.ylabel("Count")
plt.tight_layout()
plt.savefig("logs/confidence_histogram.png")
plt.close()

print("[Visualizer] Tag frequency and confidence distribution plots saved to logs/.")



import os
import yaml
import random
from pathlib import Path

CONFIG_PATH = Path("system_config.yaml")
CACHE_BASE = Path("hotcache")
```

```
class LogicRamScheduler:
    def __init__(self):
        self.shards = self.load_shards()

    def load_shards(self):
        if not CONFIG_PATH.exists():
            raise FileNotFoundError("Missing config file for logic RAM")
        with open(CONFIG_PATH, 'r') as f:
            config = yaml.safe_load(f)
        return config.get("logic_ram", {})

    def get_next_shard(self):
        if not self.shards:
            raise RuntimeError("No logic RAM shards defined")
        return random.choice(list(self.shards.values()))

    def assign_fragment(self, fragment_id, data):
        target = Path(self.get_next_shard()) / f"{fragment_id}.bin"
        with open(target, 'wb') as f:
            f.write(data)
        print(f"[RAM Scheduler] ? Assigned fragment {fragment_id} to {target}")

if __name__ == "__main__":
    scheduler = LogicRamScheduler()
    for i in range(3):
        fake_id = f"frag_{random.randint(1000, 9999)}"
        fake_data = os.urandom(2048)  # simulate 2KB fragment
        scheduler.assign_fragment(fake_id, fake_data)
```

```
  from pathlib import Path
from core.utils import load_yaml, validate_fragment, mkdir

FRAGMENTS_DIR = Path("fragments/core")
ACTIVATION_LOG = Path("logs/context_activation.log")
mkdir(ACTIVATION_LOG.parent)

class ContextActivator:
    def __init__(self, activation_threshold=0.75):
        self.threshold = activation_threshold

    def scan_fragments(self):
        activated = []
        for frag_file in FRAGMENTS_DIR.glob("*.yaml"):
            frag = load_yaml(frag_file, validate_schema=validate_fragment)
            if frag and frag.get("confidence", 0.5) >= self.threshold:
                activated.append(frag)
        return activated

    def log_activations(self, activations):
```

~~~
        with open(ACTIVATION_LOG, 'a') as log:
            for frag in activations:
                log.write(f"[ACTIVATED] {frag['id']} :: {frag.get('claim', '???')}
")
        print(f"[ContextActivator] {len(activations)} fragment(s) activated.")

    def run(self):
        active = self.scan_fragments()
        self.log_activations(active)

if __name__ == "__main__":
    ctx = ContextActivator()
    ctx.run()
```python
import yaml
import random
from pathlib import Path

FRAGMENTS_DIR = Path("fragments/core")
ACTIVATION_LOG = Path("logs/context_activation.log")
ACTIVATION_LOG.parent.mkdir(parents=True, exist_ok=True)

class ContextActivator:
    def __init__(self, activation_threshold=0.75):
        self.threshold = activation_threshold

    def scan_fragments(self):
        activated = []
        for frag_file in FRAGMENTS_DIR.glob("*.yaml"):
            try:
                with open(frag_file, 'r') as f:
                    frag = yaml.safe_load(f)
                if frag.get("confidence", 0.5) >= self.threshold:
                    activated.append(frag)
            except Exception as e:
                print(f"Error reading {frag_file.name}: {e}")
        return activated

    def log_activations(self, activations):
        with open(ACTIVATION_LOG, 'a') as log:
            for frag in activations:
                log.write(f"[ACTIVATED] {frag['id']} :: {frag.get('claim', '???')}
")
        print(f"[ContextActivator] {len(activations)} fragment(s) activated.")

    def run(self):
        active = self.scan_fragments()
        self.log_activations(active)

if __name__ == "__main__":
    ctx = ContextActivator()
    ctx.run()
~~~

```
 import shutil
import os
from pathlib import Path
import yaml

CORE_DIR = Path("fragments/core")
TARGETS = [Path("fragments/node1"), Path("fragments/node2")]
TRANSFER_LOG = Path("logs/teleport_log.txt")
TRANSFER_LOG.parent.mkdir(parents=True, exist_ok=True)

# Ensure targets exist
for target in TARGETS:
    target.mkdir(parents=True, exist_ok=True)

class FragmentTeleporter:
    def __init__(self, limit=5):
        self.limit = limit

    def select_fragments(self):
        frags = list(CORE_DIR.glob("*.yaml"))
        return frags[:self.limit] if frags else []

    def teleport(self):
        selections = self.select_fragments()
        for i, frag_path in enumerate(selections):
            target = TARGETS[i % len(TARGETS)] / frag_path.name
            shutil.move(str(frag_path), target)
            with open(TRANSFER_LOG, 'a') as log:
                log.write(f"[TELEPORTED] {frag_path.name} ? {target}
")
            print(f"[Teleporter] {frag_path.name} ? {target}")

if __name__ == "__main__":
    teleporter = FragmentTeleporter(limit=10)
    teleporter.teleport()
```

```
 import asyncio
import random
import yaml
from pathlib import Path

AGENT_DIR = Path("agents")
AGENT_DIR.mkdir(exist_ok=True)

# Dummy async task
async def swarm_worker(agent_id, delay_range=(1, 5)):
    await asyncio.sleep(random.uniform(*delay_range))
    print(f"[Swarm] Agent {agent_id} activated.")
    return agent_id

async def launch_swarm(agent_count=8):
    tasks = []
    for i in range(agent_count):
        aid = f"agent_{i+1:03}"
        tasks.append(swarm_worker(aid))

    results = await asyncio.gather(*tasks)
    log_path = Path("logs/swarm_boot.log")
    log_path.parent.mkdir(parents=True, exist_ok=True)

    with open(log_path, 'a') as log:
        for agent in results:
            log.write(f"[BOOTED] {agent}
")

    print(f"[Swarm] Launched {len(results)} agents.")

if __name__ == "__main__":
    asyncio.run(launch_swarm(agent_count=6))
```

```
 import os
import yaml
from pathlib import Path

LAYER_MAP_PATH = Path("subcon_map.yaml")
FRAGMENTS_DIR = Path("fragments/core")
OUTPUT_PATH = Path("meta/subcon_layer_cache.yaml")
OUTPUT_PATH.parent.mkdir(parents=True, exist_ok=True)

class SubconLayerMapper:
    def __init__(self):
        self.layer_map = self.load_map()

    def load_map(self):
        if not LAYER_MAP_PATH.exists():
            print("[Mapper] No layer map found. Returning empty.")
            return {}
        with open(LAYER_MAP_PATH, 'r') as f:
            return yaml.safe_load(f)

    def extract_links(self):
        results = {}
        for file in FRAGMENTS_DIR.glob("*.yaml"):
            try:
                with open(file, 'r') as f:
                    frag = yaml.safe_load(f)
                tags = frag.get("tags", [])
                for tag in tags:
                    if tag in self.layer_map:
                        results.setdefault(tag, []).append(frag['id'])
            except Exception as e:
                print(f"[Mapper] Failed to read {file.name}: {e}")
        return results

    def save_cache(self, data):
        with open(OUTPUT_PATH, 'w') as out:
            yaml.dump(data, out)
        print(f"[Mapper] Saved subcon layer associations ? {OUTPUT_PATH}")

    def run(self):
        links = self.extract_links()
        self.save_cache(links)

if __name__ == "__main__":
    mapper = SubconLayerMapper()
    mapper.run()
```

```
 import time
import uuid
import random
from pathlib import Path
from core.utils import load_yaml, save_yaml, validate_fragment, generate_uuid, timestamp
from core.cortex_bus import send_message
import yaml

FRAG_DIR = Path("fragments/core")
LOG_PATH = Path("logs/mutation_log.txt")
LOG_PATH.parent.mkdir(parents=True, exist_ok=True)

class MutationEngine:
    def __init__(self, agent_id="mutation_engine_01"):
        self.agent_id = agent_id

    def decay_confidence(self, frag):
        current = frag.get("confidence", 0.5)
        decay = 0.01 + random.uniform(0.005, 0.02)
        return max(0.0, current - decay)

    def mutate_claim(self, claim):
        if random.random() < 0.5:
            return f"It is possible that {claim.lower()}"
        else:
            return f"Not {claim.strip()}"
    def mutate_fragment(self, path, frag):
        new_claim = self.mutate_claim(frag['claim'])
        return {
            'id': generate_uuid(),
            'origin': str(path),
            'claim': new_claim,
            'parent_id': frag.get('id', None),
            'confidence': self.decay_confidence(frag),
            'emotion': frag.get('emotion', {}),
            'timestamp': int(time.time())
        }

    def save_mutation(self, new_frag):
        new_path = FRAG_DIR / f"{new_frag['id']}.yaml"
        save_yaml(new_frag, new_path)
        with open(LOG_PATH, 'a') as log:
            log.write(f"[{new_frag['timestamp']}] Mutation: {new_frag['id']} from {new_frag.get('parent_id')}
")
        send_message({
            'from': self.agent_id,
            'type': 'mutation_event',
            'payload': new_frag,
            'timestamp': new_frag['timestamp']
        })

    def run(self):
        for path in FRAG_DIR.glob("*.yaml"):
            frag = load_yaml(path, validate_schema=validate_fragment)
            if frag:
                mutated = self.mutate_fragment(path, frag)
                self.save_mutation(mutated)
                time.sleep(0.1)

if __name__ == "__main__":
    MutationEngine().run()
```

```
 import time
import random
from pathlib import Path
from core.utils import load_yaml, validate_fragment, timestamp
from core.cortex_bus import send_message

FRAG_DIR = Path("fragments/core")
LOG_PATH = Path("logs/dreamwalker_log.txt")
LOG_PATH.parent.mkdir(parents=True, exist_ok=True)

class Dreamwalker:
    def __init__(self, agent_id="dreamwalker_01"):
        self.agent_id = agent_id
        self.visited = set()
    def recursive_walk(self, frag, depth=0, lineage=None):
        if not frag or 'claim' not in frag:
            return

        lineage = lineage or []
        lineage.append(frag['claim'])
        frag_id = frag.get('id', str(random.randint(1000, 9999)))
        if frag_id in self.visited or depth > 10:
            return

        self.visited.add(frag_id)

        send_message({
            'from': self.agent_id,
            'type': 'deep_walk_event',
            'payload': {
                'claim': frag['claim'],
                'depth': depth,
                'lineage': lineage[-3:],
                'timestamp': int(time.time())
            },
            'timestamp': int(time.time())
        })

        with open(LOG_PATH, 'a') as log:
            log.write(f"Depth {depth} :: {' -> '.join(lineage[-3:])}
")

        links = frag.get('tags', [])
        for file in FRAG_DIR.glob("*.yaml"):
            next_frag = load_yaml(file, validate_schema=validate_fragment)
            if not next_frag or next_frag.get('id') in self.visited:
                continue
            if any(tag in next_frag.get('tags', []) for tag in links):
                self.recursive_walk(next_frag, depth + 1, lineage[:])

    def run(self):
        frag_files = list(FRAG_DIR.glob("*.yaml"))
        random.shuffle(frag_files)
        for path in frag_files:
            frag = load_yaml(path, validate_schema=validate_fragment)
            if frag:
                self.recursive_walk(frag)
            time.sleep(0.1)

if __name__ == "__main__":
    Dreamwalker().run()




 import time
import random
from pathlib import Path
```

```python
from core.utils import load_yaml, save_yaml, validate_fragment, generate_uuid, timestamp
from core.cortex_bus import send_message
import yaml

SOURCE_PATH = Path("meta/seed_bank.yaml")
OUTPUT_DIR = Path("fragments/core")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

class BeliefIngestor:
    def __init__(self, agent_id="belief_ingestor_01"):
        self.agent_id = agent_id

    def run(self):
        if not SOURCE_PATH.exists():
            print("[Ingestor] No seed bank found.")
            return

        seed_bank = load_yaml(SOURCE_PATH)
        if not isinstance(seed_bank, list):
            print("[Ingestor] Invalid seed bank format.")
            return

        for entry in seed_bank:
            if 'claim' not in entry:
                continue

            new_frag = {
                'id': generate_uuid(),
                'origin': str(SOURCE_PATH),
                'claim': entry['claim'],
                'tags': entry.get('tags', ["seed"]),
                'confidence': entry.get('confidence', round(random.uniform(0.6, 0.9), 2)),
                'emotion': entry.get('emotion', {}),
                'timestamp': int(time.time())
            }

            fname = f"frag_{new_frag['id']}.yaml"
            save_yaml(new_frag, OUTPUT_DIR / fname)

            send_message({
                'from': self.agent_id,
                'type': 'belief_ingested',
                'payload': new_frag,
                'timestamp': new_frag['timestamp']
            })
            time.sleep(0.05)

if __name__ == "__main__":
    BeliefIngestor().run()
```

```
 import time
import random
from pathlib import Path
from core.utils import load_yaml, validate_fragment, save_yaml, generate_uuid
from core.cortex_bus import send_message
import yaml

INPUT_DIR = Path("meta/logic_queue")
OUTPUT_DIR = Path("fragments/core")
LOG_PATH = Path("logs/logic_scrape.log")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
LOG_PATH.parent.mkdir(parents=True, exist_ok=True)

class LogicScraper:
    def __init__(self, agent_id="scraper_01"):
        self.agent_id = agent_id

    def scan_queue(self):
        return list(INPUT_DIR.glob("*.yaml"))

    def dispatch(self, path):
        frag = load_yaml(path)
        if not frag or 'claim' not in frag:
            return

        frag['id'] = generate_uuid()
        frag['tags'] = frag.get('tags', ["scraped"])
        frag['timestamp'] = int(time.time())
        save_path = OUTPUT_DIR / f"frag_{frag['id']}.yaml"
        save_yaml(frag, save_path)

        send_message({
            'from': self.agent_id,
            'type': 'logic_scraped',
            'payload': frag,
            'timestamp': frag['timestamp']
        })

        with open(LOG_PATH, 'a') as log:
            log.write(f"[SCRAPE] {frag['id']} :: {frag['claim']}
")

        path.unlink()  # remove original

    def run(self):
        while True:
            queue = self.scan_queue()
            if not queue:
                time.sleep(1)
                continue
            for file in queue:
                self.dispatch(file)
                time.sleep(0.1)
```

```
if __name__ == "__main__":
    LogicScraper().run()
```

```
 import time
from pathlib import Path
from core.utils import load_yaml, save_yaml, generate_uuid, validate_fragment

SOURCE = Path("fragments/temp")
DEST = Path("fragments/core")
SOURCE.mkdir(parents=True, exist_ok=True)
DEST.mkdir(parents=True, exist_ok=True)

class FragmentTeleporter:
    def __init__(self, agent_id="teleporter_01"):
        self.agent_id = agent_id

    def teleport(self):
        for file in SOURCE.glob("*.yaml"):
            frag = load_yaml(file, validate_schema=validate_fragment)
            if frag:
                frag['id'] = generate_uuid()
                frag['teleported_from'] = str(file)
                frag['timestamp'] = int(time.time())
                dest_path = DEST / f"frag_{frag['id']}.yaml"
                save_yaml(frag, dest_path)
                print(f"[TP] {file.name} ? {dest_path.name}")
                file.unlink()

    def run(self):
        print("[Teleporter] Scanning temp fragments...")
        self.teleport()

if __name__ == "__main__":
    FragmentTeleporter().run()
```

```
 import time
import psutil
import random
from pathlib import Path
from core.utils import load_yaml, save_yaml, validate_fragment

SOURCE_DIR = Path("fragments/core")
CACHE_DIR = Path("runtime/ramcache")
```

```
CACHE_DIR.mkdir(parents=True, exist_ok=True)

class LogicRamScheduler:
    def __init__(self, threshold=65.0):
        self.threshold = threshold

    def ram_pressure(self):
        return psutil.virtual_memory().percent

    def select_fragments(self):
        all_files = list(SOURCE_DIR.glob("*.yaml"))
        random.shuffle(all_files)
        return all_files[:min(10, len(all_files))]

    def schedule(self):
        pressure = self.ram_pressure()
        if pressure > self.threshold:
            print(f"[RAM] Skipping load ? pressure at {pressure:.1f}%")
            return

        for file in self.select_fragments():
            frag = load_yaml(file, validate_schema=validate_fragment)
            if frag:
                dest = CACHE_DIR / file.name
                save_yaml(frag, dest)
                print(f"[RAM] Cached fragment: {file.name}")

    def run(self):
        while True:
            self.schedule()
            time.sleep(5)

if __name__ == "__main__":
    LogicRamScheduler().run()
```

```
 import time
import psutil
import random
from pathlib import Path
from core.utils import load_yaml, save_yaml, validate_fragment

SOURCE_DIR = Path("fragments/core")
CACHE_DIR = Path("runtime/ramcache")
CACHE_DIR.mkdir(parents=True, exist_ok=True)

class LogicRamScheduler:
    def __init__(self, threshold=65.0):
        self.threshold = threshold

    def ram_pressure(self):
        return psutil.virtual_memory().percent
```