```
    # print(n_id, f_mapper)

    # print(f"Forwarding {n_id}")
    if self.neurons[n_id].predecessor_neurons:
        for pred in self.neurons[n_id].predecessor_neurons:
            # print(f"Predecessor {pred} = {self.neurons[pred].value}")
            self.neurons[n_id].value = self.neurons[n_id].value + (
                self.weights[(pred, n_id)] * self.neurons[pred].value
            )
            # print(f"New Value: {n_id} = {self.neurons[n_id].value}")
            self.neurons[n_id].value = f_mapper[n_id] * self.neurons[
                n_id
            ].activation_fn(self.neurons[n_id].value)
    else:
        self.neurons[n_id].value = f_mapper[n_id]
    if type(self.neurons[n_id].value) == torch.Tensor:
        queue.put((n_id, self.neurons[n_id].value.detach()))
    else:
        queue.put((n_id, self.neurons[n_id].value))
    # print(f"FINAL: {n_id} = {self.neurons[n_id].value}")

def fast_forward(self, f_mapper):
    """Fast forward the network with the given inputs"""
    if not self.is_fresh:
        raise Exception(
            "Network has already been forwarded. You may want to reset it."
        )
    self.has_forwarded = True
    self.is_fresh = False

    layer_mapper = [[] for _ in range(self.num_layers)]
    for n_id in self.topo_order:
        layer_mapper[self.neurons[n_id].layer].append(n_id)

    manager = mp.Manager()
    queue = manager.Queue()

    # pool_tuple =

    # print(layer_mapper)
    pool = Pool(mp.cpu_count())
    for layer in range(self.num_layers):
        pool.starmap(
            self._forward_layer,
            zip(layer_mapper[layer], repeat(f_mapper), repeat(queue)),
        )
        while not queue.empty():
            n_id, value = queue.get()
            # print(f"{n_id} = {value}")
            self.neurons[n_id].value = value
    pool.close()
    pool.join()

    return torch.stack([self.neurons[i].value for i in self.output_neurons])
```

```
    def forward(self, f_mapper):
        if not self.is_fresh:
            raise Exception(
                "Network has already been forwarded. You may want to reset it."
            )
        self.has_forwarded = True
        self.is_fresh = False
        for n_id in self.topo_order:
            if self.neurons[n_id].predecessor_neurons:
                for pred in self.neurons[n_id].predecessor_neurons:
                    self.neurons[n_id].value = self.neurons[n_id].value + (
                        self.weights[(pred, n_id)] * self.neurons[pred].value
                    )
                self.neurons[n_id].value = f_mapper[n_id] * self.neurons[
                    n_id
                ].activation_fn(self.neurons[n_id].value)
            else:
                self.neurons[n_id].value = f_mapper[n_id]

        return torch.stack([self.neurons[i].value for i in self.output_neurons])

    def parameters(self):
        return (p for p in self.weights.values())

    def set_neuron_activation(
        self,
        n_id: int,
        activation_fn: Callable,
    ):
        self.neurons[n_id].set_activation_fn(activation_fn)

    def reset(self):
        for n in self.neurons:
            n.value = 0
            n.grad = 0
            n.relevance = 0
        self.is_fresh = True

    def to(self, device):
                                                                                                                                                #
https://discuss.pytorch.org/t/tensor-to-device-changes-is-leaf-causing-cant-optimize-a-non-leaf-tensor/37659
        for key, value in self.weights.items():
            self.weights[key] = (
                self.weights[key].to(device).detach().requires_grad_(True)
            )

    def get_weights(self):
        return self.weights

    def set_weights(self, weights):
        self.weights = weights

    def get_gradients(self):
```

```
    grads = []
    for p in self.parameters():
        grad = None if p.grad is None else p.grad.data.cpu().numpy()
        grads.append(grad)
    return grads

def set_gradients(self, gradients):
    for g, p in zip(gradients, self.parameters()):
        if g is not None:
            p.grad = torch.from_numpy(g)

def _set_output_neurons(self):
    self.output_neurons = []
    for n in self.neurons:
        if len(n.successor_neurons) == 0:
            self.output_neurons.append(n.n_id)

def _setup_neurons(self):
    rev_adj_list = [[] for _ in range(self.num_neurons)]
    for i in range(self.num_neurons):
        for e in self.adj_list[i]:
            rev_adj_list[e].append(i)
    for u in self.neurons:
        u.set_successor_neurons(self.adj_list[u.n_id])
    for v in self.neurons:
        v.set_predecessor_neurons(rev_adj_list[v.n_id])

def _set_weights(self):
    """
    This function sets the weights of the network.
    """
    weights = {}
    for u in range(self.num_neurons):
        for v in self.adj_list[u]:
            weights[(u, v)] = torch.rand(1, requires_grad=True)
    return weights

def _topological_sort_util(self, v, visited, stack):
    # Mark the current node as visited.
    visited[v] = True

    # Recur for all the vertices adjacent to this vertex
    for i in self.adj_list[v]:
        if visited[i] is False:
            self._topological_sort_util(i, visited, stack)

    # Push current vertex to stack which stores result
    stack.append(v)

def _topological_sort(self):
    """Returns the topological sorted order of a graph"""
    visited = [False] * self.num_neurons
    stack = []
    for i in range(self.num_neurons):
```

```
            if visited[i] is False:
                self._topological_sort_util(i, visited, stack)
        return stack[::-1]

    def _assign_layers(self):
        """Assigns layers to neurons of the network"""
        for n in self.neurons:
            if len(n.predecessor_neurons) == 0:
                n.layer = 0

        for n_id in self.topo_order:
            if len(self.neurons[n_id].predecessor_neurons) > 0:
                self.neurons[n_id].layer = (
                    max(
                        [
                            self.neurons[i].layer
                            for i in self.neurons[n_id].predecessor_neurons
                        ]
                    )
                    + 1
                )
                self.num_layers = max(self.num_layers, self.neurons[n_id].layer + 1)

    def lrp(self, R, n_id):
        for n in self.neurons:
            if n.relevance != 0:
                raise Exception("Relevances are not cleared, try reseting the network")
        self.neurons[n_id].relevance = R
        for n_id in self.topo_order[::-1]:
            for succ in self.neurons[n_id].successor_neurons:
                my_contribution = 1e-9
                total_contribution = 1e-9
                for pred in self.neurons[succ].predecessor_neurons:
                    if pred == n_id:
                        my_contribution = (
                            self.neurons[n_id].value * self.weights[(pred, succ)]
                        )
                        total_contribution += my_contribution
                    else:
                        total_contribution += (
                            self.neurons[pred].value * self.weights[(pred, succ)]
                        )

                self.neurons[n_id].relevance += (
                    self.neurons[succ].relevance * my_contribution / total_contribution
                )

==== neuro_lock.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: neuro_lock.py
Purpose: Freeze current belief + memory state into a snapshot archive for backup or audit
"""
```

```
import os
import tarfile
import time
import threading
from utils import agent_profiler
# [PROFILER_INJECTED]
threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()
from pathlib import Path
import shutil

SNAPSHOT_DIR = Path("snapshots")
SNAPSHOT_DIR.mkdir(parents=True, exist_ok=True)

TARGETS = [
    "fragments/core",
    "logs/",
    "core/cortex_memory.db",
    "fragments/archive",
    "fragments/overflow"
]

LOCK_FILE = Path("core/neuro.lock")

def create_snapshot():
    timestamp = int(time.time())
    filename = SNAPSHOT_DIR / f"logicshredder_snapshot_{timestamp}.tar.gz"
    with tarfile.open(filename, "w:gz") as tar:
        for target in TARGETS:
            path = Path(target)
            if path.exists():
                tar.add(str(path), arcname=path.name)
                print(f"[neuro_lock] Added to archive: {path}")
    print(f"[neuro_lock] Snapshot complete -> {filename.name}")
    return filename

def lock_brain():
    with open(LOCK_FILE, 'w', encoding='utf-8') as lock:
        lock.write(str(int(time.time())))
    print("[neuro_lock] LOGICSHREDDER locked ? no mutations will be accepted.")

def unlock_brain():
    if LOCK_FILE.exists():
        LOCK_FILE.unlink()
        print("[neuro_lock] Brain unlocked ? mutation resumed.")

def is_locked():
    return LOCK_FILE.exists()

def toggle_lock():
    if is_locked():
        unlock_brain()
    else:
        lock_brain()
```

```
if __name__ == "__main__":
    print("\nLOGICSHREDDER :: SNAPSHOT + FREEZE SYSTEM")
    print("1. Create snapshot")
    print("2. Toggle lock/unlock")
    print("3. Check lock status")
    choice = input("\nSelect an action: ").strip()

    if choice == "1":
        create_snapshot()
    elif choice == "2":
        toggle_lock()
    elif choice == "3":
        if is_locked():
            print("? Brain is currently LOCKED.")
        else:
            print("INFO Brain is currently ACTIVE.")
    else:
        print("Invalid choice.")
# [CONFIG_PATCHED]

==== neuron.py ====
from typing import Callable

import torch
import torch.nn.functional as F


class Neuron:
    def __init__(
        self,
        n_id: int,
        activation_fn: Callable = F.relu,
    ):
        self.activation_fn = activation_fn
        self.n_id = n_id
        self.value = torch.tensor(0)
        self.grad = 0
        self.relevance = 0
        self.layer = 0
        self.predecessor_neurons = []
        self.successor_neurons = []

    def set_successor_neurons(self, successor_neurons: list):
        self.successor_neurons = successor_neurons

    def set_predecessor_neurons(self, predecessor_neurons: list):
        self.predecessor_neurons = predecessor_neurons

    def set_activation_fn(self, activation_fn: Callable):
        self.activation_fn = activation_fn

    def __repr__(self):
        return (
            super().__repr__()
```

```
            + f"""\n{self.n_id}: {self.value}\t Grad: {self.grad}
            \nPredecessor: {self.predecessor_neurons}\tSuccessor: {self.successor_neurons}"""
        )

    def __str__(self):
        return f"""\n{self.n_id}: {self.value}\t Grad: {self.grad}
            \nPredecessor: {self.predecessor_neurons}\tSuccessor: {self.successor_neurons}"""

==== nvme_memory_shim.py ====
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"

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]):
        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"[OK] Config updated with NVMe logic cache: {list(zones.values())}")

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

==== param_server.py ====
import numpy as np
import ray
import torch

from crm.core import Network


@ray.remote
class ParameterServer(object):
    def __init__(self, lr, num_neurons, adj_list, custom_activations=None):
        self.network = Network(num_neurons, adj_list, custom_activations)
        self.optimizer = torch.optim.SGD(self.network.parameters(), lr=lr)

    def apply_gradients(self, *gradients):
        summed_gradients = [
            np.stack(gradient_zip).sum(axis=0) for gradient_zip in zip(*gradients)
        ]
        self.optimizer.zero_grad()
        self.network.set_gradients(summed_gradients)
        self.optimizer.step()
        return self.network.get_weights()

    def get_weights(self):
        return self.network.get_weights()

==== patch_agents_config.py ====
"""
LOGICSHREDDER :: patch_agents_config.py
Purpose: Inject config_loader usage into all agents and core modules
"""

from pathlib import Path
import shutil

TARGET_DIRS = ["agents", "core"]
```

```
BACKUP_DIR = Path("patch_backups")
BACKUP_DIR.mkdir(parents=True, exist_ok=True)

CONFIG_IMPORT = "from core.config_loader import get"
PATCH_TAG = "# [CONFIG_PATCHED]"

patch_targets = {
    "decay = ": "decay = get('tuning.decay_rate', 0.03)",
    "CONFIDENCE_THRESHOLD = ": "CONFIDENCE_THRESHOLD = get('tuning.cold_logic_threshold', 0.3)",
    "mutation_aggression = ": "mutation_aggression = get('tuning.mutation_aggression', 0.7)",
    "curiosity_bias = ": "curiosity_bias = get('tuning.curiosity_bias', 0.4)",
    "contradiction_sensitivity = ": "contradiction_sensitivity = get('tuning.contradiction_sensitivity', 0.8)",
                    "if     Path(\"core/neuro.lock\").exists()":     "if     get('brain.lock_respect')     and
Path(\"core/neuro.lock\").exists()"
}

def patch_file(path):
    try:
        text = path.read_text(encoding='utf-8')
        if PATCH_TAG in text:
            print(f"[patch] Skipped: {path.name} (already patched)")
            return

        backup_path = BACKUP_DIR / path.name
        shutil.copy(path, backup_path)

        if CONFIG_IMPORT not in text:
            text = CONFIG_IMPORT + "\n" + text

        for old, new in patch_targets.items():
            text = text.replace(old, new)

        text += f"\n{PATCH_TAG}\n"
        path.write_text(text, encoding='utf-8')
        print(f"[patch] [OK] Patched: {path.name}")

    except Exception as e:
        print(f"[patch] ERROR Error on {path.name}: {e}")

def main():
    print("INFO Beginning config patch sweep...")
    for dir_name in TARGET_DIRS:
        path = Path(dir_name)
        if not path.exists():
            print(f"[patch] Directory not found: {dir_name}")
            continue

        for file in path.glob("*.py"):
            patch_file(file)
    print("[OK] Patch complete.")

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

```
==== patch_auto_configurator_disks.py ====
"""
LOGICSHREDDER :: patch_auto_configurator_disks.py
Purpose: Patch auto_configurator.py with multi-disk detection logic
"""

from pathlib import Path
import shutil

TARGET = Path("auto_configurator.py")
BACKUP = Path("auto_configurator.backup.py")

PATCH_FUNC = '''
def get_system_profile():
    disks = []
    for part in psutil.disk_partitions():
        try:
            usage = psutil.disk_usage(part.mountpoint)
            disks.append({
                "mount": part.mountpoint,
                "fstype": part.fstype,
                "free_gb": round(usage.free / (1024**3), 2),
                "total_gb": round(usage.total / (1024**3), 2)
            })
        except PermissionError:
            continue

    total_free = sum(d['free_gb'] for d in disks)
    primary = disks[0] if disks else {"mount": "?", "free_gb": 0}

    return {
        "cores": psutil.cpu_count(logical=False),
        "threads": psutil.cpu_count(logical=True),
        "total_ram": round(psutil.virtual_memory().total / (1024**2)),
        "available_ram": round(psutil.virtual_memory().available / (1024**2)),
        "disks": disks,
        "disk_free_total": round(total_free, 2),
        "disk_primary_mount": primary["mount"],
        "platform": platform.system()
    }
'''

def patch_file():
    if not TARGET.exists():
        print("[patch] ERROR File not found: auto_configurator.py")
        return

    shutil.copy(TARGET, BACKUP)
    print("[patch] ? Backup saved as:", BACKUP.name)

    code = TARGET.read_text(encoding='utf-8')
    start = code.find("def get_system_profile()")
    end = code.find("def", start + 10)
```

```
    if start == -1:
        print("[patch] ERROR Could not find original function.")
        return

    # Replace old function
    new_code = code[:start] + PATCH_FUNC + "\n" + code[end:]
    TARGET.write_text(new_code, encoding='utf-8')

    print("[patch] [OK] auto_configurator.py successfully patched with multi-disk awareness.")

if __name__ == "__main__":
    patch_file()

==== patch_config_references.py ====
"""
LOGICSHREDDER :: patch_config_references.py
Purpose: Replace old core.config_loader references with core.config_access
"""

from pathlib import Path
import shutil

BASE_DIR = Path(".")
BACKUP_DIR = Path("patch_backups_config_access")
BACKUP_DIR.mkdir(exist_ok=True)

TARGET = "core.config_loader"
REPLACEMENT = "core.config_access"

def patch_file(path):
    try:
        code = path.read_text(encoding='utf-8')
        if TARGET not in code or REPLACEMENT in code:
            return False

        backup_path = BACKUP_DIR / path.name
        shutil.copy(path, backup_path)

        patched = code.replace(TARGET, REPLACEMENT)
        path.write_text(patched, encoding='utf-8')
        print(f"[patch] [OK] Patched: {path}")
        return True
    except Exception as e:
        print(f"[patch] ERROR Error patching {path}: {e}")
        return False

def main():
    print("[patch] ? Searching for config_loader references...")
    for file in BASE_DIR.rglob("*.py"):
        patch_file(file)
    print("[patch] ? Config access references updated.")

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

```
==== patch_diskfree_totaltotal.py ====
"""
LOGICSHREDDER :: patch_diskfree_totaltotal.py
Purpose: Fix accidental 'disk_free_total_total' typo in auto_configurator.py
"""

from pathlib import Path
import shutil

target = Path("auto_configurator.py")
backup = Path("auto_configurator.repaired.py")

if not target.exists():
    print("ERROR auto_configurator.py not found.")
    exit(1)

code = target.read_text(encoding="utf-8")

if "disk_free_total_total" not in code:
    print("[OK] No 'disk_free_total_total' found. Already clean.")
    exit(0)

shutil.copy(target, backup)
print(f"? Backup saved as {backup.name}")

# FIX THE TYPED ABOMINATION
patched = code.replace("disk_free_total_total", "disk_free_total")
target.write_text(patched, encoding="utf-8")

print("[OK] Fixed: 'disk_free_total_total' -> 'disk_free_total'")
print("? Logic has been purified. Run the boot again.")

==== patch_diskfree_typo.py ====
"""
LOGICSHREDDER :: patch_diskfree_typo.py
Purpose: Replace all old 'disk_free' keys with 'disk_free_total' in auto_configurator.py
"""

from pathlib import Path
import shutil

target = Path("auto_configurator.py")
backup = Path("auto_configurator.backup2.py")

if not target.exists():
    print("ERROR auto_configurator.py not found.")
    exit(1)

text = target.read_text(encoding="utf-8")

if "disk_free" not in text:
    print("[OK] No 'disk_free' references left. You're clean.")
    exit(0)
```

```
# Make a backup
shutil.copy(target, backup)
print(f"? Backup saved as {backup.name}")

# Replace all occurrences
patched = text.replace("disk_free", "disk_free_total")
target.write_text(patched, encoding="utf-8")

print("[OK] All 'disk_free' references replaced with 'disk_free_total'")
print("? You are now free of the circular logic hemorrhage.")

==== patch_symbolic_emitters.py ====
# patch_symbolic_emitters.py
import os
from pathlib import Path

PATCHES = {
    "validator.py": {
        "inject": "import redis\nr = redis.Redis(decode_responses=True)\n",
        "target": "send_message({",
        "payload": "r.publish(\"contradiction_found\", payload['claim_1'])  # [AUTO_EMIT]"
    },
    "mutation_engine.py": {
        "inject": "import redis\nr = redis.Redis(decode_responses=True)\n",
        "target": "send_message({",
        "payload": "r.publish(\"decay_event\", new_frag['claim'])  # [AUTO_EMIT]"
    },
    "dreamwalker.py": {
        "inject": "import redis\nr = redis.Redis(decode_responses=True)\n",
        "target": "send_message({",
          "payload": "if frag.get('confidence', 1.0) < 0.4 and depth > 5:\n     r.publish(\"symbolic_alert\",
frag['claim'])  # [AUTO_EMIT]"
    }
}

def patch_file(filename, inject_code, hook_line, emit_line):
    path = Path(filename)
    if not path.exists():
        print(f"[!] Skipped missing file: {filename}")
        return

    lines = path.read_text(encoding="utf-8").splitlines()
    modified = []
    injected = False
    hooked = False

    for line in lines:
        if not injected and "import" in line and "yaml" in line:
            modified.append(line)
            modified.append(inject_code)
            injected = True
        elif not hooked and hook_line in line:
            modified.append(emit_line)
```

```
            modified.append(line)
            hooked = True
        else:
            modified.append(line)

    path.write_text("\n".join(modified), encoding="utf-8")
    print(f"[?] Patched {filename}")

if __name__ == "__main__":
    for file, cfg in PATCHES.items():
        patch_file(file, cfg["inject"], cfg["target"], cfg["payload"])

==== path_optimizer.py ====
"""
LOGICSHREDDER :: path_optimizer.py
Purpose: Benchmark all mounted drives and assign best paths in config
"""

import psutil
import time
import os
import tempfile
from pathlib import Path
import yaml

CONFIG_PATH = Path("configs/system_config.yaml")
BENCH_FILE = "shredspeed.tmp"

def benchmark_disk(mountpoint, duration=1.5):
    path = Path(mountpoint) / BENCH_FILE
    block = b"x" * 4096
    try:
        # Write test
        start = time.perf_counter()
        with open(path, 'wb') as f:
            while time.perf_counter() - start < duration:
                f.write(block)
        write_time = time.perf_counter() - start

        # Read test
        start = time.perf_counter()
        with open(path, 'rb') as f:
            while f.read(4096):
                pass
        read_time = time.perf_counter() - start

        ops = round((duration / write_time + duration / read_time) / 2, 2)
        path.unlink(missing_ok=True)
        return ops
    except Exception as e:
        print(f"[path_optimizer] ERROR Skipped {mountpoint}: {e}")
        return 0.0

def find_best_disks():
```

```
    candidates = []
    for part in psutil.disk_partitions(all=False):
        try:
            if "cdrom" in part.opts.lower() or part.fstype == "":
                continue
            usage = psutil.disk_usage(part.mountpoint)
            speed = benchmark_disk(part.mountpoint)
            candidates.append({
                "mount": part.mountpoint,
                "free_gb": round(usage.free / (1024**3), 2),
                "speed_score": speed
            })
        except:
            continue
    return sorted(candidates, key=lambda x: x["speed_score"], reverse=True)

def write_path_config(disks):
    if not disks:
        print("[path_optimizer] No valid disks found.")
        return

    paths = {
        "fragments": Path(disks[0]["mount"]) / "logicshred/fragments/core/",
        "archive": Path(disks[-1]["mount"]) / "logicshred/fragments/archive/",
        "cold": Path(disks[-1]["mount"]) / "logicshred/fragments/cold/",
        "overflow": Path(disks[-1]["mount"]) / "logicshred/fragments/overflow/",
        "logs": Path(disks[1 if len(disks) > 1 else 0]["mount"]) / "logicshred/logs/",
        "profiler": Path(disks[1 if len(disks) > 1 else 0]["mount"]) / "logicshred/logs/agent_stats/",
        "snapshot": Path(disks[-1]["mount"]) / "logicshred/snapshots/",
        "input": Path(disks[0]["mount"]) / "logicshred/input/"
    }

    if CONFIG_PATH.exists():
        with open(CONFIG_PATH, 'r', encoding='utf-8') as f:
            config = yaml.safe_load(f)
    else:
        config = {}

    config['paths'] = {k: str(v).replace("\\", "/") for k, v in paths.items()}
    config['brain']['optimized_by'] = "path_optimizer"

    with open(CONFIG_PATH, 'w', encoding='utf-8') as f:
        yaml.dump(config, f)

    print("[path_optimizer] [OK] Config paths updated for optimal storage.")

def main():
    print("[path_optimizer] LAUNCH Scanning all drives...")
    disks = find_best_disks()
    for d in disks:
        print(f"  {d['mount']} -> {d['speed_score']} ops/sec | Free: {d['free_gb']} GB")
    write_path_config(disks)

if __name__ == "__main__":
```

```
    main()

==== plot_parameter_server.py ====
"""
Parameter Server
================

The parameter server is a framework for distributed machine learning training.

In the parameter server framework, a centralized server (or group of server
nodes) maintains global shared parameters of a machine-learning model
(e.g., a neural network) while the data and computation of calculating
updates (i.e., gradient descent updates) are distributed over worker nodes.

.. image:: /ray-core/images/param_actor.png
    :align: center
z
Parameter servers are a core part of many machine learning applications. This
document walks through how to implement simple synchronous and asynchronous
parameter servers using Ray actors.

To run the application, first install some dependencies.

.. code-block:: bash

  pip install torch torchvision filelock

Let's first define some helper functions and import some dependencies.

"""
import os
import time

import numpy as np
import ray
import torch
import torch.nn as nn
import torch.nn.functional as F
from filelock import FileLock
from torchvision import datasets, transforms


def get_data_loader():
    """Safely downloads data. Returns training/validation set dataloader."""
    mnist_transforms = transforms.Compose(
        [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
    )

    # We add FileLock here because multiple workers will want to
    # download data, and this may cause overwrites since
    # DataLoader is not threadsafe.
    with FileLock(os.path.expanduser("~/data.lock")):
        train_loader = torch.utils.data.DataLoader(
            datasets.MNIST(
```

```
                "~/data", train=True, download=True, transform=mnist_transforms
            ),
            batch_size=128,
            shuffle=True,
        )
        test_loader = torch.utils.data.DataLoader(
            datasets.MNIST("~/data", train=False, transform=mnist_transforms),
            batch_size=128,
            shuffle=True,
        )
    return train_loader, test_loader


def evaluate(model, test_loader):
    """Evaluates the accuracy of the model on a validation dataset."""
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(test_loader):
            # This is only set to finish evaluation faster.
            if batch_idx * len(data) > 1024:
                break
            outputs = model(data)
            _, predicted = torch.max(outputs.data, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
    return 100.0 * correct / total


######################################################################
# Setup: Defining the Neural Network
# ----------------------------------
#
# We define a small neural network to use in training. We provide
# some helper functions for obtaining data, including getter/setter
# methods for gradients and weights.


class ConvNet(nn.Module):
    """Small ConvNet for MNIST."""

    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 3, kernel_size=3)
        self.fc = nn.Linear(192, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 3))
        x = x.view(-1, 192)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)

    def get_weights(self):
```

```
        return {k: v.cpu() for k, v in self.state_dict().items()}

    def set_weights(self, weights):
        self.load_state_dict(weights)

    def get_gradients(self):
        grads = []
        for p in self.parameters():
            grad = None if p.grad is None else p.grad.data.cpu().numpy()
            grads.append(grad)
        return grads

    def set_gradients(self, gradients):
        for g, p in zip(gradients, self.parameters()):
            if g is not None:
                p.grad = torch.from_numpy(g)


##########################################################################
# Defining the Parameter Server
# -----------------------------
#
# The parameter server will hold a copy of the model.
# During training, it will:
#
# 1. Receive gradients and apply them to its model.
#
# 2. Send the updated model back to the workers.
#
# The ``@ray.remote`` decorator defines a remote process. It wraps the
# ParameterServer class and allows users to instantiate it as a
# remote actor.


@ray.remote
class ParameterServer(object):
    def __init__(self, lr):
        self.model = ConvNet()
        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=lr)

    def apply_gradients(self, *gradients):
        summed_gradients = [
            np.stack(gradient_zip).sum(axis=0) for gradient_zip in zip(*gradients)
        ]
        self.optimizer.zero_grad()
        self.model.set_gradients(summed_gradients)
        self.optimizer.step()
        return self.model.get_weights()

    def get_weights(self):
        return self.model.get_weights()


##########################################################################
```

```
# Defining the Worker
# -------------------
# The worker will also hold a copy of the model. During training. it will
# continuously evaluate data and send gradients
# to the parameter server. The worker will synchronize its model with the
# Parameter Server model weights.


@ray.remote
class DataWorker(object):
    def __init__(self):
        self.model = ConvNet()
        self.data_iterator = iter(get_data_loader()[0])

    def compute_gradients(self, weights):
        self.model.set_weights(weights)
        try:
            data, target = next(self.data_iterator)
        except StopIteration:  # When the epoch ends, start a new epoch.
            self.data_iterator = iter(get_data_loader()[0])
            data, target = next(self.data_iterator)
        self.model.zero_grad()
        output = self.model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        return self.model.get_gradients()


iterations = 250
num_workers = 10

######################################################################
# We'll also instantiate a model on the driver process to evaluate the test
# accuracy during training.

model = ConvNet()
test_loader = get_data_loader()[1]


######################################################################
# Asynchronous Parameter Server Training
# --------------------------------------
# We'll now create a synchronous parameter server training scheme. We'll first
# instantiate a process for the parameter server, along with multiple
# workers.


start_time = time.time()

print("Running Asynchronous Parameter Server Training.")

ray.init(ignore_reinit_error=True)
ps = ParameterServer.remote(1e-2)
workers = [DataWorker.remote() for i in range(num_workers)]
```

```
############################################################################
# Here, workers will asynchronously compute the gradients given its
# current weights and send these gradients to the parameter server as
# soon as they are ready. When the Parameter server finishes applying the
# new gradient, the server will send back a copy of the current weights to the
# worker. The worker will then update the weights and repeat.

current_weights = ps.get_weights.remote()

gradients = {}
for worker in workers:
    gradients[worker.compute_gradients.remote(current_weights)] = worker

for i in range(iterations * num_workers):
    ready_gradient_list, _ = ray.wait(list(gradients))
    ready_gradient_id = ready_gradient_list[0]
    worker = gradients.pop(ready_gradient_id)

    # Compute and apply gradients.
    current_weights = ps.apply_gradients.remote(*[ready_gradient_id])
    gradients[worker.compute_gradients.remote(current_weights)] = worker

    if i % 10 == 0:
        # Evaluate the current model after every 10 updates.
        model.set_weights(ray.get(current_weights))
        accuracy = evaluate(model, test_loader)
        print("Iter {}: \taccuracy is {:.1f}".format(i, accuracy))

print("Final accuracy is {:.1f}.".format(accuracy))
end_time = time.time()
print("Time taken: {}".format(end_time - start_time))
############################################################################################
# Final Thoughts
# --------------
#
# This approach is powerful because it enables you to implement a parameter
# server with a few lines of code as part of a Python application.
# As a result, this simplifies the deployment of applications that use
# parameter servers and to modify the behavior of the parameter server.
#
# For example, sharding the parameter server, changing the update rule,
# switching between asynchronous and synchronous updates, ignoring
# straggler workers, or any number of other customizations,
# will only require a few extra lines of code.

==== pop.py ====
import os
import requests
from tqdm import tqdm

# === CONFIG ===
DOWNLOAD_DIR = "massive_datasets"
os.makedirs(DOWNLOAD_DIR, exist_ok=True)
```

```
# === LIST OF DATASETS ===
DATASETS = {
    # --- TEXT ---
    "pile_train": "https://the-eye.eu/public/AI/pile/train.jsonl.zst",
    "pile_val":   "https://the-eye.eu/public/AI/pile/val.jsonl.zst",
    "pile_test":  "https://the-eye.eu/public/AI/pile/test.jsonl.zst",
    "wikipedia_en": "https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2",
                                                                                "commoncrawl_sample":
"https://data.commoncrawl.org/crawl-data/CC-MAIN-2024-10/segments/1700071270090.54/wet/CC-MAIN-20240318003555-2
0240318033555-00000.warc.wet.gz",

    # --- IMAGE ---
    "coco_train2017": "http://images.cocodataset.org/zips/train2017.zip",
    "coco_val2017":   "http://images.cocodataset.org/zips/val2017.zip",
    "coco_ann":       "http://images.cocodataset.org/annotations/annotations_trainval2017.zip",
    "openimages_classes": "https://storage.googleapis.com/openimages/2018_04/class-descriptions-boxable.csv",

    # --- AUDIO ---
    "librispeech_dev": "https://www.openslr.org/resources/12/dev-clean.tar.gz",
    "librispeech_test": "https://www.openslr.org/resources/12/test-clean.tar.gz",
    "librispeech_train_100": "https://www.openslr.org/resources/12/train-clean-100.tar.gz",
    "librispeech_train_360": "https://www.openslr.org/resources/12/train-clean-360.tar.gz",
    "librispeech_train_other": "https://www.openslr.org/resources/12/train-other-500.tar.gz",

    # --- VIDEO ---
    "ucf101": "https://www.crcv.ucf.edu/data/UCF101/UCF101.rar",

    # --- BIO / SCI ---
    # GenBank is FTP only ? we skip it or handle separately
}

# === DOWNLOAD FUNC ===
def download(url, filename):
    path = os.path.join(DOWNLOAD_DIR, filename)
    try:
        with requests.get(url, stream=True, timeout=60) as r:
            r.raise_for_status()
            total = int(r.headers.get('content-length', 0))
            with open(path, 'wb') as f, tqdm(
                total=total,
                unit='B',
                unit_scale=True,
                unit_divisor=1024,
                desc=filename
            ) as bar:
                for chunk in r.iter_content(1024 * 1024):
                    if chunk:
                        f.write(chunk)
                        bar.update(len(chunk))
        print(f"[OK] Downloaded: {filename}")
    except Exception as e:
        print(f"[?] Failed: {filename}\n    URL: {url}\n    Reason: {e}")
```

```
# === GO ===
for name, url in DATASETS.items():
    filename = url.split("/")[-1]
    download(url, filename)

print("\n[OK] All downloads attempted. Check the 'massive_datasets' folder.")

==== pydantic_models_to_grammar.py ====
from __future__ import annotations

import inspect
import json
import re
from copy import copy
from enum import Enum
from inspect import getdoc, isclass
from typing import TYPE_CHECKING, Any, Callable, List, Optional, Union, get_args, get_origin, get_type_hints

from docstring_parser import parse
from pydantic import BaseModel, create_model

if TYPE_CHECKING:
    from types import GenericAlias
else:
    # python 3.8 compat
    from typing import _GenericAlias as GenericAlias

# TODO: fix this
# pyright: reportAttributeAccessIssue=information


class PydanticDataType(Enum):
    """
    Defines the data types supported by the grammar_generator.

    Attributes:
        STRING (str): Represents a string data type.
        BOOLEAN (str): Represents a boolean data type.
        INTEGER (str): Represents an integer data type.
        FLOAT (str): Represents a float data type.
        OBJECT (str): Represents an object data type.
        ARRAY (str): Represents an array data type.
        ENUM (str): Represents an enum data type.
        CUSTOM_CLASS (str): Represents a custom class data type.
    """

    STRING = "string"
    TRIPLE_QUOTED_STRING = "triple_quoted_string"
    MARKDOWN_CODE_BLOCK = "markdown_code_block"
    BOOLEAN = "boolean"
    INTEGER = "integer"
    FLOAT = "float"
    OBJECT = "object"
    ARRAY = "array"
```

```
    ENUM = "enum"
    ANY = "any"
    NULL = "null"
    CUSTOM_CLASS = "custom-class"
    CUSTOM_DICT = "custom-dict"
    SET = "set"


def map_pydantic_type_to_gbnf(pydantic_type: type[Any]) -> str:
    origin_type = get_origin(pydantic_type)
    origin_type = pydantic_type if origin_type is None else origin_type

    if isclass(origin_type) and issubclass(origin_type, str):
        return PydanticDataType.STRING.value
    elif isclass(origin_type) and issubclass(origin_type, bool):
        return PydanticDataType.BOOLEAN.value
    elif isclass(origin_type) and issubclass(origin_type, int):
        return PydanticDataType.INTEGER.value
    elif isclass(origin_type) and issubclass(origin_type, float):
        return PydanticDataType.FLOAT.value
    elif isclass(origin_type) and issubclass(origin_type, Enum):
        return PydanticDataType.ENUM.value

    elif isclass(origin_type) and issubclass(origin_type, BaseModel):
        return format_model_and_field_name(origin_type.__name__)
    elif origin_type is list:
        element_type = get_args(pydantic_type)[0]
        return f"{map_pydantic_type_to_gbnf(element_type)}-list"
    elif origin_type is set:
        element_type = get_args(pydantic_type)[0]
        return f"{map_pydantic_type_to_gbnf(element_type)}-set"
    elif origin_type is Union:
        union_types = get_args(pydantic_type)
        union_rules = [map_pydantic_type_to_gbnf(ut) for ut in union_types]
        return f"union-{'-or-'.join(union_rules)}"
    elif origin_type is Optional:
        element_type = get_args(pydantic_type)[0]
        return f"optional-{map_pydantic_type_to_gbnf(element_type)}"
    elif isclass(origin_type):
        return f"{PydanticDataType.CUSTOM_CLASS.value}-{format_model_and_field_name(origin_type.__name__)}"
    elif origin_type is dict:
        key_type, value_type = get_args(pydantic_type)
                                                                                    return
f"custom-dict-key-type-{format_model_and_field_name(map_pydantic_type_to_gbnf(key_type))}-value-type-{format_mo
del_and_field_name(map_pydantic_type_to_gbnf(value_type))}"
    else:
        return "unknown"


def format_model_and_field_name(model_name: str) -> str:
    parts = re.findall("[A-Z][^A-Z]*", model_name)
    if not parts:  # Check if the list is empty
        return model_name.lower().replace("_", "-")
    return "-".join(part.lower().replace("_", "-") for part in parts)
```

```
def generate_list_rule(element_type):
    """
    Generate a GBNF rule for a list of a given element type.

    :param element_type: The type of the elements in the list (e.g., 'string').
    :return: A string representing the GBNF rule for a list of the given type.
    """
    rule_name = f"{map_pydantic_type_to_gbnf(element_type)}-list"
    element_rule = map_pydantic_type_to_gbnf(element_type)
    list_rule = rf'{rule_name} ::= "["  {element_rule} ("," {element_rule})* "]"'
    return list_rule


def get_members_structure(cls, rule_name):
    if issubclass(cls, Enum):
        # Handle Enum types
        members = [f'"\\"{member.value}\\""' for name, member in cls.__members__.items()]
        return f"{cls.__name__.lower()} ::= " + " | ".join(members)
    if cls.__annotations__ and cls.__annotations__ != {}:
        result = f'{rule_name} ::= "{{"'
        # Modify this comprehension
        members = [
            f'  "\\"{name}\\"" ":"  {map_pydantic_type_to_gbnf(param_type)}'
            for name, param_type in get_type_hints(cls).items()
            if name != "self"
        ]

        result += '"," '.join(members)
        result += '  "}"'
        return result
    if rule_name == "custom-class-any":
        result = f"{rule_name} ::= "
        result += "value"
        return result

    init_signature = inspect.signature(cls.__init__)
    parameters = init_signature.parameters
    result = f'{rule_name} ::=  "{{"'
    # Modify this comprehension too
    members = [
        f'  "\\"{name}\\"" ":"  {map_pydantic_type_to_gbnf(param.annotation)}'
        for name, param in parameters.items()
        if name != "self" and param.annotation != inspect.Parameter.empty
    ]

    result += '", "'.join(members)
    result += '  "}"'
    return result


def regex_to_gbnf(regex_pattern: str) -> str:
    """
```

```
    Translate a basic regex pattern to a GBNF rule.
    Note: This function handles only a subset of simple regex patterns.
    """
    gbnf_rule = regex_pattern

    # Translate common regex components to GBNF
    gbnf_rule = gbnf_rule.replace("\\d", "[0-9]")
    gbnf_rule = gbnf_rule.replace("\\s", "[ \t\n]")

    # Handle quantifiers and other regex syntax that is similar in GBNF
    # (e.g., '*', '+', '?', character classes)

    return gbnf_rule


def generate_gbnf_integer_rules(max_digit=None, min_digit=None):
    """

    Generate GBNF Integer Rules

     Generates GBNF (Generalized Backus-Naur Form) rules for integers based on the given maximum and minimum digits.

    Parameters:
        max_digit (int): The maximum number of digits for the integer. Default is None.
        min_digit (int): The minimum number of digits for the integer. Default is None.

    Returns:
        integer_rule (str): The identifier for the integer rule generated.
         additional_rules (list): A list of additional rules generated based on the given maximum and minimum digits.

    """
    additional_rules = []

    # Define the rule identifier based on max_digit and min_digit
    integer_rule = "integer-part"
    if max_digit is not None:
        integer_rule += f"-max{max_digit}"
    if min_digit is not None:
        integer_rule += f"-min{min_digit}"

    # Handling Integer Rules
    if max_digit is not None or min_digit is not None:
        # Start with an empty rule part
        integer_rule_part = ""

        # Add mandatory digits as per min_digit
        if min_digit is not None:
            integer_rule_part += "[0-9] " * min_digit

        # Add optional digits up to max_digit
        if max_digit is not None:
            optional_digits = max_digit - (min_digit if min_digit is not None else 0)
```

```
            integer_rule_part += "".join(["[0-9]? " for _ in range(optional_digits)])

        # Trim the rule part and append it to additional rules
        integer_rule_part = integer_rule_part.strip()
        if integer_rule_part:
            additional_rules.append(f"{integer_rule} ::= {integer_rule_part}")

    return integer_rule, additional_rules


def generate_gbnf_float_rules(max_digit=None, min_digit=None, max_precision=None, min_precision=None):
    """
    Generate GBNF float rules based on the given constraints.

    :param max_digit: Maximum number of digits in the integer part (default: None)
    :param min_digit: Minimum number of digits in the integer part (default: None)
    :param max_precision: Maximum number of digits in the fractional part (default: None)
    :param min_precision: Minimum number of digits in the fractional part (default: None)
    :return: A tuple containing the float rule and additional rules as a list

    Example Usage:
    max_digit = 3
    min_digit = 1
    max_precision = 2
    min_precision = 1
    generate_gbnf_float_rules(max_digit, min_digit, max_precision, min_precision)

    Output:
     ('float-3-1-2-1', ['integer-part-max3-min1 ::= [0-9] [0-9] [0-9]?', 'fractional-part-max2-min1 ::= [0-9]
[0-9]?', 'float-3-1-2-1 ::= integer-part-max3-min1 "." fractional-part-max2-min
    *1'])

    Note:
     GBNF stands for Generalized Backus-Naur Form, which is a notation technique to specify the syntax of
programming languages or other formal grammars.
    """
    additional_rules = []

    # Define the integer part rule
    integer_part_rule = (
        "integer-part"
        + (f"-max{max_digit}" if max_digit is not None else "")
        + (f"-min{min_digit}" if min_digit is not None else "")
    )

    # Define the fractional part rule based on precision constraints
    fractional_part_rule = "fractional-part"
    fractional_rule_part = ""
    if max_precision is not None or min_precision is not None:
        fractional_part_rule += (f"-max{max_precision}" if max_precision is not None else "") + (
            f"-min{min_precision}" if min_precision is not None else ""
        )
        # Minimum number of digits
        fractional_rule_part = "[0-9]" * (min_precision if min_precision is not None else 1)
```

```
        # Optional additional digits
        fractional_rule_part += "".join(
            [" [0-9]?"] * ((max_precision - (
                min_precision if min_precision is not None else 1)) if max_precision is not None else 0)
        )
        additional_rules.append(f"{fractional_part_rule} ::= {fractional_rule_part}")

    # Define the float rule
    float_rule = f"float-{max_digit if max_digit is not None else 'X'}-{min_digit if min_digit is not None else
'X'}-{max_precision if max_precision is not None else 'X'}-{min_precision if min_precision is not None else
'X'}"
    additional_rules.append(f'{float_rule} ::= {integer_part_rule} "." {fractional_part_rule}')

    # Generating the integer part rule definition, if necessary
    if max_digit is not None or min_digit is not None:
        integer_rule_part = "[0-9]"
        if min_digit is not None and min_digit > 1:
            integer_rule_part += " [0-9]" * (min_digit - 1)
        if max_digit is not None:
             integer_rule_part += "".join([" [0-9]?"] * (max_digit - (min_digit if min_digit is not None else
1)))
        additional_rules.append(f"{integer_part_rule} ::= {integer_rule_part.strip()}")

    return float_rule, additional_rules


def generate_gbnf_rule_for_type(
    model_name, field_name, field_type, is_optional, processed_models, created_rules, field_info=None
) -> tuple[str, list[str]]:
    """
    Generate GBNF rule for a given field type.

    :param model_name: Name of the model.

    :param field_name: Name of the field.
    :param field_type: Type of the field.
    :param is_optional: Whether the field is optional.
    :param processed_models: List of processed models.
    :param created_rules: List of created rules.
    :param field_info: Additional information about the field (optional).

    :return: Tuple containing the GBNF type and a list of additional rules.
    :rtype: tuple[str, list]
    """
    rules = []

    field_name = format_model_and_field_name(field_name)
    gbnf_type = map_pydantic_type_to_gbnf(field_type)

    origin_type = get_origin(field_type)
    origin_type = field_type if origin_type is None else origin_type

    if isclass(origin_type) and issubclass(origin_type, BaseModel):
        nested_model_name = format_model_and_field_name(field_type.__name__)
```

```
        nested_model_rules, _ = generate_gbnf_grammar(field_type, processed_models, created_rules)
        rules.extend(nested_model_rules)
        gbnf_type, rules = nested_model_name, rules
    elif isclass(origin_type) and issubclass(origin_type, Enum):
        enum_values = [f'"\\"{e.value}\\""' for e in field_type]  # Adding escaped quotes
        enum_rule = f"{model_name}-{field_name} ::= {' | '.join(enum_values)}"
        rules.append(enum_rule)
        gbnf_type, rules = model_name + "-" + field_name, rules
    elif origin_type is list:  # Array
        element_type = get_args(field_type)[0]
        element_rule_name, additional_rules = generate_gbnf_rule_for_type(
            model_name, f"{field_name}-element", element_type, is_optional, processed_models, created_rules
        )
        rules.extend(additional_rules)
        array_rule = f"""{model_name}-{field_name} ::= "[" ws {element_rule_name} ("," ws {element_rule_name})*
 "]" """
        rules.append(array_rule)
        gbnf_type, rules = model_name + "-" + field_name, rules

    elif origin_type is set:  # Array
        element_type = get_args(field_type)[0]
        element_rule_name, additional_rules = generate_gbnf_rule_for_type(
            model_name, f"{field_name}-element", element_type, is_optional, processed_models, created_rules
        )
        rules.extend(additional_rules)
        array_rule = f"""{model_name}-{field_name} ::= "[" ws {element_rule_name} ("," ws {element_rule_name})*
 "]" """
        rules.append(array_rule)
        gbnf_type, rules = model_name + "-" + field_name, rules

    elif gbnf_type.startswith("custom-class-"):
        rules.append(get_members_structure(field_type, gbnf_type))
    elif gbnf_type.startswith("custom-dict-"):
        key_type, value_type = get_args(field_type)

        additional_key_type, additional_key_rules = generate_gbnf_rule_for_type(
            model_name, f"{field_name}-key-type", key_type, is_optional, processed_models, created_rules
        )
        additional_value_type, additional_value_rules = generate_gbnf_rule_for_type(
            model_name, f"{field_name}-value-type", value_type, is_optional, processed_models, created_rules
        )
        gbnf_type = rf'{gbnf_type} ::= "{{"  ( {additional_key_type} ": "  {additional_value_type} ("," "\n" ws
{additional_key_type} ":"  {additional_value_type})*  )? "}}" '

        rules.extend(additional_key_rules)
        rules.extend(additional_value_rules)
    elif gbnf_type.startswith("union-"):
        union_types = get_args(field_type)
        union_rules = []

        for union_type in union_types:
            if isinstance(union_type, GenericAlias):
                union_gbnf_type, union_rules_list = generate_gbnf_rule_for_type(
                    model_name, field_name, union_type, False, processed_models, created_rules
```

```
                )
                union_rules.append(union_gbnf_type)
                rules.extend(union_rules_list)

            elif not issubclass(union_type, type(None)):
                union_gbnf_type, union_rules_list = generate_gbnf_rule_for_type(
                    model_name, field_name, union_type, False, processed_models, created_rules
                )
                union_rules.append(union_gbnf_type)
                rules.extend(union_rules_list)

        # Defining the union grammar rule separately
        if len(union_rules) == 1:
            union_grammar_rule = f"{model_name}-{field_name}-optional ::= {' | '.join(union_rules)} | null"
        else:
            union_grammar_rule = f"{model_name}-{field_name}-union ::= {' | '.join(union_rules)}"
        rules.append(union_grammar_rule)
        if len(union_rules) == 1:
            gbnf_type = f"{model_name}-{field_name}-optional"
        else:
            gbnf_type = f"{model_name}-{field_name}-union"
    elif isclass(origin_type) and issubclass(origin_type, str):
           if field_info and hasattr(field_info, "json_schema_extra") and field_info.json_schema_extra is not
None:
            triple_quoted_string = field_info.json_schema_extra.get("triple_quoted_string", False)
            markdown_string = field_info.json_schema_extra.get("markdown_code_block", False)

                        gbnf_type = PydanticDataType.TRIPLE_QUOTED_STRING.value if triple_quoted_string else
PydanticDataType.STRING.value
            gbnf_type = PydanticDataType.MARKDOWN_CODE_BLOCK.value if markdown_string else gbnf_type

        elif field_info and hasattr(field_info, "pattern"):
            # Convert regex pattern to grammar rule
            regex_pattern = field_info.regex.pattern
            gbnf_type = f"pattern-{field_name} ::= {regex_to_gbnf(regex_pattern)}"
        else:
            gbnf_type = PydanticDataType.STRING.value

    elif (
        isclass(origin_type)
        and issubclass(origin_type, float)
        and field_info
        and hasattr(field_info, "json_schema_extra")
        and field_info.json_schema_extra is not None
    ):
        # Retrieve precision attributes for floats
        max_precision = (
            field_info.json_schema_extra.get("max_precision") if field_info and hasattr(field_info,
                                                                                        "json_schema_extra")
else None
        )
        min_precision = (
            field_info.json_schema_extra.get("min_precision") if field_info and hasattr(field_info,
                                                                                        "json_schema_extra")
```

```
else None
        )
        max_digits = field_info.json_schema_extra.get("max_digit") if field_info and hasattr(field_info,

"json_schema_extra") else None
        min_digits = field_info.json_schema_extra.get("min_digit") if field_info and hasattr(field_info,

"json_schema_extra") else None

        # Generate GBNF rule for float with given attributes
        gbnf_type, rules = generate_gbnf_float_rules(
                                    max_digit=max_digits,   min_digit=min_digits,   max_precision=max_precision,
min_precision=min_precision
        )

    elif (
        isclass(origin_type)
        and issubclass(origin_type, int)
        and field_info
        and hasattr(field_info, "json_schema_extra")
        and field_info.json_schema_extra is not None
    ):
        # Retrieve digit attributes for integers
        max_digits = field_info.json_schema_extra.get("max_digit") if field_info and hasattr(field_info,

"json_schema_extra") else None
        min_digits = field_info.json_schema_extra.get("min_digit") if field_info and hasattr(field_info,

"json_schema_extra") else None

        # Generate GBNF rule for integer with given attributes
        gbnf_type, rules = generate_gbnf_integer_rules(max_digit=max_digits, min_digit=min_digits)
    else:
        gbnf_type, rules = gbnf_type, []

    return gbnf_type, rules


def  generate_gbnf_grammar(model:  type[BaseModel],  processed_models:  set[type[BaseModel]],  created_rules:
dict[str, list[str]]) -> tuple[list[str], bool]:
    """

    Generate GBnF Grammar

    Generates a GBnF grammar for a given model.

    :param model: A Pydantic model class to generate the grammar for. Must be a subclass of BaseModel.
    :param processed_models: A set of already processed models to prevent infinite recursion.
    :param created_rules: A dict containing already created rules to prevent duplicates.
    :return: A list of GBnF grammar rules in string format. And two booleans indicating if an extra markdown or
triple quoted string is in the grammar.
    Example Usage:
    ```
    model = MyModel
```
```

```
processed_models = set()
created_rules = dict()

gbnf_grammar = generate_gbnf_grammar(model, processed_models, created_rules)
```
"""
if model in processed_models:
    return [], False

processed_models.add(model)
model_name = format_model_and_field_name(model.__name__)

if not issubclass(model, BaseModel):
    # For non-Pydantic classes, generate model_fields from __annotations__ or __init__
    if hasattr(model, "__annotations__") and model.__annotations__:
        model_fields = {name: (typ, ...) for name, typ in get_type_hints(model).items()}
    else:
        init_signature = inspect.signature(model.__init__)
        parameters = init_signature.parameters
        model_fields = {name: (param.annotation, param.default) for name, param in parameters.items() if
                        name != "self"}
else:
    # For Pydantic models, use model_fields and check for ellipsis (required fields)
    model_fields = get_type_hints(model)

model_rule_parts = []
nested_rules = []
has_markdown_code_block = False
has_triple_quoted_string = False
look_for_markdown_code_block = False
look_for_triple_quoted_string = False
for field_name, field_info in model_fields.items():
    if not issubclass(model, BaseModel):
        field_type, default_value = field_info
        # Check if the field is optional (not required)
        is_optional = (default_value is not inspect.Parameter.empty) and (default_value is not Ellipsis)
    else:
        field_type = field_info
        field_info = model.model_fields[field_name]
        is_optional = field_info.is_required is False and get_origin(field_type) is Optional
    rule_name, additional_rules = generate_gbnf_rule_for_type(
        model_name, format_model_and_field_name(field_name), field_type, is_optional, processed_models,
        created_rules, field_info
)
    look_for_markdown_code_block = True if rule_name == "markdown_code_block" else False
    look_for_triple_quoted_string = True if rule_name == "triple_quoted_string" else False
    if not look_for_markdown_code_block and not look_for_triple_quoted_string:
        if rule_name not in created_rules:
            created_rules[rule_name] = additional_rules
        model_rule_parts.append(f' ws "\\"{field_name}\\"" ":" ws {rule_name}') # Adding escaped quotes
        nested_rules.extend(additional_rules)
    else:
        has_triple_quoted_string = look_for_triple_quoted_string
        has_markdown_code_block = look_for_markdown_code_block
```
```

```
    fields_joined = r' "," "\n" '.join(model_rule_parts)
    model_rule = rf'{model_name} ::= "{{" "\n" {fields_joined} "\n" ws "}}"'

    has_special_string = False
    if has_triple_quoted_string:
        model_rule += '"\\n" ws "}"'
        model_rule += '"\\n" triple-quoted-string'
        has_special_string = True
    if has_markdown_code_block:
        model_rule += '"\\n" ws "}"'
        model_rule += '"\\n" markdown-code-block'
        has_special_string = True
    all_rules = [model_rule] + nested_rules

    return all_rules, has_special_string


def generate_gbnf_grammar_from_pydantic_models(
     models: list[type[BaseModel]], outer_object_name: str | None = None, outer_object_content: str | None = None,
    list_of_outputs: bool = False
) -> str:
    """
    Generate GBNF Grammar from Pydantic Models.

    This method takes a list of Pydantic models and uses them to generate a GBNF grammar string. The generated grammar string can be used for parsing and validating data using the generated
    * grammar.

    Args:
        models (list[type[BaseModel]]): A list of Pydantic models to generate the grammar from.
          outer_object_name (str): Outer object name for the GBNF grammar. If None, no outer object will be generated. Eg. "function" for function calling.
         outer_object_content (str): Content for the outer rule in the GBNF grammar. Eg. "function_parameters" or "params" for function calling.
        list_of_outputs (str, optional): Allows a list of output objects
    Returns:
        str: The generated GBNF grammar string.

    Examples:
        models = [UserModel, PostModel]
        grammar = generate_gbnf_grammar_from_pydantic(models)
        print(grammar)
        # Output:
        # root ::= UserModel | PostModel
        # ...
    """
    processed_models: set[type[BaseModel]] = set()
    all_rules = []
    created_rules: dict[str, list[str]] = {}
    if outer_object_name is None:
        for model in models:
            model_rules, _ = generate_gbnf_grammar(model, processed_models, created_rules)
```

```
            all_rules.extend(model_rules)

        if list_of_outputs:
            root_rule = r'root ::= (" "| "\n") "[" ws grammar-models ("," ws grammar-models)* ws "]"' + "\n"
        else:
            root_rule = r'root ::= (" "| "\n") grammar-models' + "\n"
        root_rule += "grammar-models ::= " + " | ".join(
            [format_model_and_field_name(model.__name__) for model in models])
        all_rules.insert(0, root_rule)
        return "\n".join(all_rules)
    elif outer_object_name is not None:
        if list_of_outputs:
            root_rule = (
                    rf'root ::= (" "| "\n") "[" ws {format_model_and_field_name(outer_object_name)} ("," ws
{format_model_and_field_name(outer_object_name)})* ws "]"'
                + "\n"
            )
        else:
            root_rule = f"root ::= {format_model_and_field_name(outer_object_name)}\n"

        model_rule = (
                            rf'{format_model_and_field_name(outer_object_name)}  ::=  ("  "|  "\n")  "{{"  ws
"\"{outer_object_name}\""  ":" ws grammar-models'
        )

        fields_joined = " | ".join(
            [rf"{format_model_and_field_name(model.__name__)}-grammar-model" for model in models])

        grammar_model_rules = f"\ngrammar-models ::= {fields_joined}"
        mod_rules = []
        for model in models:
            mod_rule = rf"{format_model_and_field_name(model.__name__)}-grammar-model ::= "
            mod_rule += (
                                    rf'"\"{model.__name__}\""  ","  ws  "\"{outer_object_content}\""  ":"  ws
{format_model_and_field_name(model.__name__)}' + "\n"
            )
            mod_rules.append(mod_rule)
        grammar_model_rules += "\n" + "\n".join(mod_rules)

        for model in models:
            model_rules, has_special_string = generate_gbnf_grammar(model, processed_models,
                                                                    created_rules)

            if not has_special_string:
                model_rules[0] += r'"\n" ws "}"'

            all_rules.extend(model_rules)

        all_rules.insert(0, root_rule + model_rule + grammar_model_rules)
        return "\n".join(all_rules)


def get_primitive_grammar(grammar):
    """
```

```
    Returns the needed GBNF primitive grammar for a given GBNF grammar string.

    Args:
        grammar (str): The string containing the GBNF grammar.

    Returns:
        str: GBNF primitive grammar string.
    """
    type_list: list[type[object]] = []
    if "string-list" in grammar:
        type_list.append(str)
    if "boolean-list" in grammar:
        type_list.append(bool)
    if "integer-list" in grammar:
        type_list.append(int)
    if "float-list" in grammar:
        type_list.append(float)
    additional_grammar = [generate_list_rule(t) for t in type_list]
    primitive_grammar = r"""
boolean ::= "true" | "false"
null ::= "null"
string ::= "\"" (
        [^"\\] |
        "\\" (["\\/bfnrt] | "u" [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
      )* "\"" ws
ws ::= ([ \t\n] ws)?
float ::= ("-"? ([0] | [1-9] [0-9]*)) ("." [0-9]+)? ([eE] [-+]? [0-9]+)? ws

integer ::= [0-9]+"""

    any_block = ""
    if "custom-class-any" in grammar:
        any_block = """
value ::= object | array | string | number | boolean | null

object ::=
  "{" ws (
            string ":" ws value
    ("," ws string ":" ws value)*
  )? "}" ws

array  ::=
  "[" ws (
            value
    ("," ws value)*
  )? "]" ws

number ::= integer | float"""

    markdown_code_block_grammar = ""
    if "markdown-code-block" in grammar:
        markdown_code_block_grammar = r'''
markdown-code-block ::= opening-triple-ticks markdown-code-block-content closing-triple-ticks
markdown-code-block-content ::= ( [^`] | "`" [^`] |  "`"  "`" [^`]  )*
```

```
opening-triple-ticks ::= "```" "python" "\n" | "```" "c" "\n" | "```" "cpp" "\n" | "```" "txt" "\n" | "```"
"text" "\n" | "```" "json" "\n" | "```" "javascript" "\n" | "```" "css" "\n" | "```" "html" "\n" | "```"
"markdown" "\n"
closing-triple-ticks ::= "```" "\n"'''

    if "triple-quoted-string" in grammar:
        markdown_code_block_grammar = r"""
triple-quoted-string ::= triple-quotes triple-quoted-string-content triple-quotes
triple-quoted-string-content ::= ( [^'] | "'" [^'] |  "'"  "'" [^']  )*
triple-quotes ::= "'''" """
    return "\n" + "\n".join(additional_grammar) + any_block + primitive_grammar + markdown_code_block_grammar


def generate_markdown_documentation(
    pydantic_models: list[type[BaseModel]], model_prefix="Model", fields_prefix="Fields",
    documentation_with_field_description=True
) -> str:
    """
    Generate markdown documentation for a list of Pydantic models.

    Args:
        pydantic_models (list[type[BaseModel]]): list of Pydantic model classes.
        model_prefix (str): Prefix for the model section.
        fields_prefix (str): Prefix for the fields section.
        documentation_with_field_description (bool): Include field descriptions in the documentation.

    Returns:
        str: Generated text documentation.
    """
    documentation = ""
    pyd_models: list[tuple[type[BaseModel], bool]] = [(model, True) for model in pydantic_models]
    for model, add_prefix in pyd_models:
        if add_prefix:
            documentation += f"{model_prefix}: {model.__name__}\n"
        else:
            documentation += f"Model: {model.__name__}\n"

        # Handling multi-line model description with proper indentation

        class_doc = getdoc(model)
        base_class_doc = getdoc(BaseModel)
        class_description = class_doc if class_doc and class_doc != base_class_doc else ""
        if class_description != "":
            documentation += "  Description: "
            documentation += format_multiline_description(class_description, 0) + "\n"

        if add_prefix:
            # Indenting the fields section
            documentation += f"  {fields_prefix}:\n"
        else:
            documentation += f"  Fields:\n"  # noqa: F541
        if isclass(model) and issubclass(model, BaseModel):
            for name, field_type in get_type_hints(model).items():
                # if name == "markdown_code_block":
```

```
                #    continue
                if get_origin(field_type) == list:
                    element_type = get_args(field_type)[0]
                    if isclass(element_type) and issubclass(element_type, BaseModel):
                        pyd_models.append((element_type, False))
                if get_origin(field_type) == Union:
                    element_types = get_args(field_type)
                    for element_type in element_types:
                        if isclass(element_type) and issubclass(element_type, BaseModel):
                            pyd_models.append((element_type, False))
                documentation += generate_field_markdown(
                                                                                name,    field_type,    model,
documentation_with_field_description=documentation_with_field_description
                )
            documentation += "\n"

        if hasattr(model, "Config") and hasattr(model.Config,
                                                                            "json_schema_extra") and "example" in
model.Config.json_schema_extra:
            documentation += f"  Expected Example Output for {format_model_and_field_name(model.__name__)}:\n"
            json_example = json.dumps(model.Config.json_schema_extra["example"])
            documentation += format_multiline_description(json_example, 2) + "\n"

    return documentation


def generate_field_markdown(
    field_name: str, field_type: type[Any], model: type[BaseModel], depth=1,
    documentation_with_field_description=True
) -> str:
    """
    Generate markdown documentation for a Pydantic model field.

    Args:
        field_name (str): Name of the field.
        field_type (type[Any]): Type of the field.
        model (type[BaseModel]): Pydantic model class.
        depth (int): Indentation depth in the documentation.
        documentation_with_field_description (bool): Include field descriptions in the documentation.

    Returns:
        str: Generated text documentation for the field.
    """
    indent = "    " * depth

    field_info = model.model_fields.get(field_name)
    field_description = field_info.description if field_info and field_info.description else ""

    origin_type = get_origin(field_type)
    origin_type = field_type if origin_type is None else origin_type

    if origin_type == list:
        element_type = get_args(field_type)[0]
                 field_text  =  f"{indent}{field_name}  ({format_model_and_field_name(field_type.__name__)}  of
```

```
{format_model_and_field_name(element_type.__name__)})"
        if field_description != "":
            field_text += ":\n"
        else:
            field_text += "\n"
    elif origin_type == Union:
        element_types = get_args(field_type)
        types = []
        for element_type in element_types:
            types.append(format_model_and_field_name(element_type.__name__))
        field_text = f"{indent}{field_name} ({' or '.join(types)})"
        if field_description != "":
            field_text += ":\n"
        else:
            field_text += "\n"
    else:
        field_text = f"{indent}{field_name} ({format_model_and_field_name(field_type.__name__)})"
        if field_description != "":
            field_text += ":\n"
        else:
            field_text += "\n"

    if not documentation_with_field_description:
        return field_text

    if field_description != "":
        field_text += f"        Description: {field_description}\n"

    # Check for and include field-specific examples if available
    if hasattr(model, "Config") and hasattr(model.Config,
                                                                        "json_schema_extra") and "example" in
model.Config.json_schema_extra:
        field_example = model.Config.json_schema_extra["example"].get(field_name)
        if field_example is not None:
            example_text = f"'{field_example}'" if isinstance(field_example, str) else field_example
            field_text += f"{indent}  Example: {example_text}\n"

    if isclass(origin_type) and issubclass(origin_type, BaseModel):
        field_text += f"{indent}  Details:\n"
        for name, type_ in get_type_hints(field_type).items():
            field_text += generate_field_markdown(name, type_, field_type, depth + 2)

    return field_text


def format_json_example(example: dict[str, Any], depth: int) -> str:
    """
    Format a JSON example into a readable string with indentation.

    Args:
        example (dict): JSON example to be formatted.
        depth (int): Indentation depth.

    Returns:
```

```
        str: Formatted JSON example string.
    """
    indent = "    " * depth
    formatted_example = "{\n"
    for key, value in example.items():
        value_text = f"'{value}'" if isinstance(value, str) else value
        formatted_example += f"{indent}{key}: {value_text},\n"
    formatted_example = formatted_example.rstrip(",\n") + "\n" + indent + "}"
    return formatted_example


def generate_text_documentation(
    pydantic_models: list[type[BaseModel]], model_prefix="Model", fields_prefix="Fields",
    documentation_with_field_description=True
) -> str:
    """
    Generate text documentation for a list of Pydantic models.

    Args:
        pydantic_models (list[type[BaseModel]]): List of Pydantic model classes.
        model_prefix (str): Prefix for the model section.
        fields_prefix (str): Prefix for the fields section.
        documentation_with_field_description (bool): Include field descriptions in the documentation.

    Returns:
        str: Generated text documentation.
    """
    documentation = ""
    pyd_models: list[tuple[type[BaseModel], bool]] = [(model, True) for model in pydantic_models]
    for model, add_prefix in pyd_models:
        if add_prefix:
            documentation += f"{model_prefix}: {model.__name__}\n"
        else:
            documentation += f"Model: {model.__name__}\n"

        # Handling multi-line model description with proper indentation

        class_doc = getdoc(model)
        base_class_doc = getdoc(BaseModel)
        class_description = class_doc if class_doc and class_doc != base_class_doc else ""
        if class_description != "":
            documentation += "  Description: "
            documentation += "\n" + format_multiline_description(class_description, 2) + "\n"

        if isclass(model) and issubclass(model, BaseModel):
            documentation_fields = ""
            for name, field_type in get_type_hints(model).items():
                # if name == "markdown_code_block":
                #    continue
                if get_origin(field_type) == list:
                    element_type = get_args(field_type)[0]
                    if isclass(element_type) and issubclass(element_type, BaseModel):
                        pyd_models.append((element_type, False))
                if get_origin(field_type) == Union:
```

```
                    element_types = get_args(field_type)
                    for element_type in element_types:
                        if isclass(element_type) and issubclass(element_type, BaseModel):
                            pyd_models.append((element_type, False))
                documentation_fields += generate_field_text(
                                                                        name,    field_type,    model,
documentation_with_field_description=documentation_with_field_description
                )
            if documentation_fields != "":
                if add_prefix:
                    documentation += f"  {fields_prefix}:\n{documentation_fields}"
                else:
                    documentation += f"  Fields:\n{documentation_fields}"
            documentation += "\n"

        if hasattr(model, "Config") and hasattr(model.Config,
                                                                    "json_schema_extra") and "example" in
model.Config.json_schema_extra:
            documentation += f"  Expected Example Output for {format_model_and_field_name(model.__name__)}:\n"
            json_example = json.dumps(model.Config.json_schema_extra["example"])
            documentation += format_multiline_description(json_example, 2) + "\n"

    return documentation


def generate_field_text(
    field_name: str, field_type: type[Any], model: type[BaseModel], depth=1,
    documentation_with_field_description=True
) -> str:
    """
    Generate text documentation for a Pydantic model field.

    Args:
        field_name (str): Name of the field.
        field_type (type[Any]): Type of the field.
        model (type[BaseModel]): Pydantic model class.
        depth (int): Indentation depth in the documentation.
        documentation_with_field_description (bool): Include field descriptions in the documentation.

    Returns:
        str: Generated text documentation for the field.
    """
    indent = "    " * depth

    field_info = model.model_fields.get(field_name)
    field_description = field_info.description if field_info and field_info.description else ""

    if get_origin(field_type) == list:
        element_type = get_args(field_type)[0]
                field_text  =  f"{indent}{field_name}  ({format_model_and_field_name(field_type.__name__)}  of
{format_model_and_field_name(element_type.__name__)})"
        if field_description != "":
            field_text += ":\n"
        else:
```

```
            field_text += "\n"
    elif get_origin(field_type) == Union:
        element_types = get_args(field_type)
        types = []
        for element_type in element_types:
            types.append(format_model_and_field_name(element_type.__name__))
        field_text = f"{indent}{field_name} ({' or '.join(types)})"
        if field_description != "":
            field_text += ":\n"
        else:
            field_text += "\n"
    else:
        field_text = f"{indent}{field_name} ({format_model_and_field_name(field_type.__name__)})"
        if field_description != "":
            field_text += ":\n"
        else:
            field_text += "\n"

    if not documentation_with_field_description:
        return field_text

    if field_description != "":
        field_text += f"{indent}  Description: " + field_description + "\n"

    # Check for and include field-specific examples if available
    if hasattr(model, "Config") and hasattr(model.Config,
                                                                        "json_schema_extra") and "example" in
model.Config.json_schema_extra:
        field_example = model.Config.json_schema_extra["example"].get(field_name)
        if field_example is not None:
            example_text = f"'{field_example}'" if isinstance(field_example, str) else field_example
            field_text += f"{indent}  Example: {example_text}\n"

    if isclass(field_type) and issubclass(field_type, BaseModel):
        field_text += f"{indent}  Details:\n"
        for name, type_ in get_type_hints(field_type).items():
            field_text += generate_field_text(name, type_, field_type, depth + 2)

    return field_text


def format_multiline_description(description: str, indent_level: int) -> str:
    """
    Format a multiline description with proper indentation.

    Args:
        description (str): Multiline description.
        indent_level (int): Indentation level.

    Returns:
        str: Formatted multiline description.
    """
    indent = "    " * indent_level
    return indent + description.replace("\n", "\n" + indent)
```

```
def save_gbnf_grammar_and_documentation(
                                    grammar,         documentation,         grammar_file_path="./grammar.gbnf",
documentation_file_path="./grammar_documentation.md"
):
    """
    Save GBNF grammar and documentation to specified files.

    Args:
        grammar (str): GBNF grammar string.
        documentation (str): Documentation string.
        grammar_file_path (str): File path to save the GBNF grammar.
        documentation_file_path (str): File path to save the documentation.

    Returns:
        None
    """
    try:
        with open(grammar_file_path, "w") as file:
            file.write(grammar + get_primitive_grammar(grammar))
        print(f"Grammar successfully saved to {grammar_file_path}")
    except IOError as e:
        print(f"An error occurred while saving the grammar file: {e}")

    try:
        with open(documentation_file_path, "w") as file:
            file.write(documentation)
        print(f"Documentation successfully saved to {documentation_file_path}")
    except IOError as e:
        print(f"An error occurred while saving the documentation file: {e}")


def remove_empty_lines(string):
    """
    Remove empty lines from a string.

    Args:
        string (str): Input string.

    Returns:
        str: String with empty lines removed.
    """
    lines = string.splitlines()
    non_empty_lines = [line for line in lines if line.strip() != ""]
    string_no_empty_lines = "\n".join(non_empty_lines)
    return string_no_empty_lines


def generate_and_save_gbnf_grammar_and_documentation(
    pydantic_model_list,
    grammar_file_path="./generated_grammar.gbnf",
    documentation_file_path="./generated_grammar_documentation.md",
    outer_object_name: str | None = None,
```

```
    outer_object_content: str | None = None,
    model_prefix: str = "Output Model",
    fields_prefix: str = "Output Fields",
    list_of_outputs: bool = False,
    documentation_with_field_description=True,
):
    """
    Generate GBNF grammar and documentation, and save them to specified files.

    Args:
        pydantic_model_list: List of Pydantic model classes.
        grammar_file_path (str): File path to save the generated GBNF grammar.
        documentation_file_path (str): File path to save the generated documentation.
          outer_object_name (str): Outer object name for the GBNF grammar. If None, no outer object will be generated. Eg. "function" for function calling.
         outer_object_content (str): Content for the outer rule in the GBNF grammar. Eg. "function_parameters" or "params" for function calling.
        model_prefix (str): Prefix for the model section in the documentation.
        fields_prefix (str): Prefix for the fields section in the documentation.
        list_of_outputs (bool): Whether the output is a list of items.
        documentation_with_field_description (bool): Include field descriptions in the documentation.

    Returns:
        None
    """
    documentation = generate_markdown_documentation(
        pydantic_model_list, model_prefix, fields_prefix,
        documentation_with_field_description=documentation_with_field_description
    )
            grammar   =   generate_gbnf_grammar_from_pydantic_models(pydantic_model_list,   outer_object_name, outer_object_content,
                                                                     list_of_outputs)
    grammar = remove_empty_lines(grammar)
    save_gbnf_grammar_and_documentation(grammar, documentation, grammar_file_path, documentation_file_path)


def generate_gbnf_grammar_and_documentation(
    pydantic_model_list,
    outer_object_name: str | None = None,
    outer_object_content: str | None = None,
    model_prefix: str = "Output Model",
    fields_prefix: str = "Output Fields",
    list_of_outputs: bool = False,
    documentation_with_field_description=True,
):
    """
    Generate GBNF grammar and documentation for a list of Pydantic models.

    Args:
        pydantic_model_list: List of Pydantic model classes.
          outer_object_name (str): Outer object name for the GBNF grammar. If None, no outer object will be generated. Eg. "function" for function calling.
         outer_object_content (str): Content for the outer rule in the GBNF grammar. Eg. "function_parameters" or "params" for function calling.
```

```
        model_prefix (str): Prefix for the model section in the documentation.
        fields_prefix (str): Prefix for the fields section in the documentation.
        list_of_outputs (bool): Whether the output is a list of items.
        documentation_with_field_description (bool): Include field descriptions in the documentation.

    Returns:
        tuple: GBNF grammar string, documentation string.
    """
    documentation = generate_markdown_documentation(
        copy(pydantic_model_list), model_prefix, fields_prefix,
        documentation_with_field_description=documentation_with_field_description
    )
            grammar   =   generate_gbnf_grammar_from_pydantic_models(pydantic_model_list,   outer_object_name,
outer_object_content,
                                                          list_of_outputs)
    grammar = remove_empty_lines(grammar + get_primitive_grammar(grammar))
    return grammar, documentation


def generate_gbnf_grammar_and_documentation_from_dictionaries(
    dictionaries: list[dict[str, Any]],
    outer_object_name: str | None = None,
    outer_object_content: str | None = None,
    model_prefix: str = "Output Model",
    fields_prefix: str = "Output Fields",
    list_of_outputs: bool = False,
    documentation_with_field_description=True,
):
    """
    Generate GBNF grammar and documentation from a list of dictionaries.

    Args:
        dictionaries (list[dict]): List of dictionaries representing Pydantic models.
          outer_object_name (str): Outer object name for the GBNF grammar. If None, no outer object will be
generated. Eg. "function" for function calling.
         outer_object_content (str): Content for the outer rule in the GBNF grammar. Eg. "function_parameters"
or "params" for function calling.
        model_prefix (str): Prefix for the model section in the documentation.
        fields_prefix (str): Prefix for the fields section in the documentation.
        list_of_outputs (bool): Whether the output is a list of items.
        documentation_with_field_description (bool): Include field descriptions in the documentation.

    Returns:
        tuple: GBNF grammar string, documentation string.
    """
    pydantic_model_list = create_dynamic_models_from_dictionaries(dictionaries)
    documentation = generate_markdown_documentation(
        copy(pydantic_model_list), model_prefix, fields_prefix,
        documentation_with_field_description=documentation_with_field_description
    )
            grammar   =   generate_gbnf_grammar_from_pydantic_models(pydantic_model_list,   outer_object_name,
outer_object_content,
                                                          list_of_outputs)
    grammar = remove_empty_lines(grammar + get_primitive_grammar(grammar))
```

```
    return grammar, documentation


def create_dynamic_model_from_function(func: Callable[..., Any]):
    """
     Creates a dynamic Pydantic model from a given function's type hints and adds the function as a 'run'
method.

    Args:
        func (Callable): A function with type hints from which to create the model.

    Returns:
        A dynamic Pydantic model class with the provided function as a 'run' method.
    """

    # Get the signature of the function
    sig = inspect.signature(func)

    # Parse the docstring
    assert func.__doc__ is not None
    docstring = parse(func.__doc__)

    dynamic_fields = {}
    param_docs = []
    for param in sig.parameters.values():
        # Exclude 'self' parameter
        if param.name == "self":
            continue

        # Assert that the parameter has a type annotation
        if param.annotation == inspect.Parameter.empty:
            raise TypeError(f"Parameter '{param.name}' in function '{func.__name__}' lacks a type annotation")

        # Find the parameter's description in the docstring
        param_doc = next((d for d in docstring.params if d.arg_name == param.name), None)

        # Assert that the parameter has a description
        if not param_doc or not param_doc.description:
            raise ValueError(
                f"Parameter '{param.name}' in function '{func.__name__}' lacks a description in the docstring")

        # Add parameter details to the schema
        param_docs.append((param.name, param_doc))
        if param.default == inspect.Parameter.empty:
            default_value = ...
        else:
            default_value = param.default
        dynamic_fields[param.name] = (
            param.annotation if param.annotation != inspect.Parameter.empty else str, default_value)
    # Creating the dynamic model
    dynamic_model = create_model(f"{func.__name__}", **dynamic_fields)

    for name, param_doc in param_docs:
        dynamic_model.model_fields[name].description = param_doc.description
```

```
    dynamic_model.__doc__ = docstring.short_description

    def run_method_wrapper(self):
        func_args = {name: getattr(self, name) for name, _ in dynamic_fields.items()}
        return func(**func_args)

    # Adding the wrapped function as a 'run' method
    setattr(dynamic_model, "run", run_method_wrapper)
    return dynamic_model


def add_run_method_to_dynamic_model(model: type[BaseModel], func: Callable[..., Any]):
    """
    Add a 'run' method to a dynamic Pydantic model, using the provided function.

    Args:
        model (type[BaseModel]): Dynamic Pydantic model class.
        func (Callable): Function to be added as a 'run' method to the model.

    Returns:
        type[BaseModel]: Pydantic model class with the added 'run' method.
    """

    def run_method_wrapper(self):
        func_args = {name: getattr(self, name) for name in model.model_fields}
        return func(**func_args)

    # Adding the wrapped function as a 'run' method
    setattr(model, "run", run_method_wrapper)

    return model


def create_dynamic_models_from_dictionaries(dictionaries: list[dict[str, Any]]):
    """
    Create a list of dynamic Pydantic model classes from a list of dictionaries.

    Args:
        dictionaries (list[dict]): List of dictionaries representing model structures.

    Returns:
        list[type[BaseModel]]: List of generated dynamic Pydantic model classes.
    """
    dynamic_models = []
    for func in dictionaries:
        model_name = format_model_and_field_name(func.get("name", ""))
        dyn_model = convert_dictionary_to_pydantic_model(func, model_name)
        dynamic_models.append(dyn_model)
    return dynamic_models


def map_grammar_names_to_pydantic_model_class(pydantic_model_list):
    output = {}
```

```
    for model in pydantic_model_list:
        output[format_model_and_field_name(model.__name__)] = model

    return output


def json_schema_to_python_types(schema):
    type_map = {
        "any": Any,
        "string": str,
        "number": float,
        "integer": int,
        "boolean": bool,
        "array": list,
    }
    return type_map[schema]


def list_to_enum(enum_name, values):
    return Enum(enum_name, {value: value for value in values})


def convert_dictionary_to_pydantic_model(dictionary: dict[str, Any], model_name: str = "CustomModel") ->
type[Any]:
    """
    Convert a dictionary to a Pydantic model class.

    Args:
        dictionary (dict): Dictionary representing the model structure.
        model_name (str): Name of the generated Pydantic model.

    Returns:
        type[BaseModel]: Generated Pydantic model class.
    """
    fields: dict[str, Any] = {}

    if "properties" in dictionary:
        for field_name, field_data in dictionary.get("properties", {}).items():
            if field_data == "object":
                submodel = convert_dictionary_to_pydantic_model(dictionary, f"{model_name}_{field_name}")
                fields[field_name] = (submodel, ...)
            else:
                field_type = field_data.get("type", "str")

                if field_data.get("enum", []):
                    fields[field_name] = (list_to_enum(field_name, field_data.get("enum", [])), ...)
                elif field_type == "array":
                    items = field_data.get("items", {})
                    if items != {}:
                        array = {"properties": items}
                                                    array_type = convert_dictionary_to_pydantic_model(array,
f"{model_name}_{field_name}_items")
                        fields[field_name] = (List[array_type], ...)
                    else:
```

```
                    fields[field_name] = (list, ...)
            elif field_type == "object":
                submodel = convert_dictionary_to_pydantic_model(field_data, f"{model_name}_{field_name}")
                fields[field_name] = (submodel, ...)
            elif field_type == "required":
                required = field_data.get("enum", [])
                for key, field in fields.items():
                    if key not in required:
                        optional_type = fields[key][0]
                        fields[key] = (Optional[optional_type], ...)
            else:
                field_type = json_schema_to_python_types(field_type)
                fields[field_name] = (field_type, ...)
    if "function" in dictionary:
        for field_name, field_data in dictionary.get("function", {}).items():
            if field_name == "name":
                model_name = field_data
            elif field_name == "description":
                fields["__doc__"] = field_data
            elif field_name == "parameters":
                return convert_dictionary_to_pydantic_model(field_data, f"{model_name}")

    if "parameters" in dictionary:
        field_data = {"function": dictionary}
        return convert_dictionary_to_pydantic_model(field_data, f"{model_name}")
    if "required" in dictionary:
        required = dictionary.get("required", [])
        for key, field in fields.items():
            if key not in required:
                optional_type = fields[key][0]
                fields[key] = (Optional[optional_type], ...)
    custom_model = create_model(model_name, **fields)
    return custom_model

==== pydantic_models_to_grammar_examples.py ====
#!/usr/bin/env python3

"""Function calling example using pydantic models."""

from __future__ import annotations

import argparse
import datetime
import json
import logging
import textwrap
import sys
from enum import Enum
from typing import Optional, Union

import requests
from pydantic import BaseModel, Field
from pydantic_models_to_grammar import (add_run_method_to_dynamic_model, convert_dictionary_to_pydantic_model,
                                        create_dynamic_model_from_function,
```

```
generate_gbnf_grammar_and_documentation)


def create_completion(host, prompt, gbnf_grammar):
    """Calls the /completion API on llama-server.

    See
    https://github.com/ggml-org/llama.cpp/tree/HEAD/examples/server#api-endpoints
    """
          print(f"     Request:\n          Grammar:\n{textwrap.indent(gbnf_grammar,   '              ')}\n
Prompt:\n{textwrap.indent(prompt.rstrip(), '      ')}")
    headers = {"Content-Type": "application/json"}
    data = {"prompt": prompt, "grammar": gbnf_grammar}
    result = requests.post(f"http://{host}/completion", headers=headers, json=data).json()
    assert data.get("error") is None, data
    logging.info("Result: %s", result)
    content = result["content"]
    print(f"  Model: {result['model']}")
    print(f"  Result:\n{textwrap.indent(json.dumps(json.loads(content), indent=2), '    ')}")
    return content


# A function for the agent to send a message to the user.
class SendMessageToUser(BaseModel):
    """Send a message to the User."""
    chain_of_thought: str = Field(..., description="Your chain of thought while sending the message.")
    message: str = Field(..., description="Message you want to send to the user.")

    def run(self):
        print(f"SendMessageToUser: {self.message}")


def example_rce(host):
    """Minimal test case where the LLM call an arbitrary python function."""
    print("- example_rce")
    tools = [SendMessageToUser]
    gbnf_grammar, documentation = generate_gbnf_grammar_and_documentation(
        pydantic_model_list=tools, outer_object_name="function",
        outer_object_content="function_parameters", model_prefix="Function", fields_prefix="Parameters")
     system_message = "You are an advanced AI, tasked to assist the user by calling functions in JSON format.
The following are the available functions and their parameters and types:\n\n" + documentation
    user_message = "What is 42 * 42?"
                                                                                prompt                    =
f"<|im_start|>system\n{system_message}<|im_end|>\n<|im_start|>user\n{user_message}<|im_end|>\n<|im_start|>assis
tant"
    text = create_completion(host, prompt, gbnf_grammar)
    json_data = json.loads(text)
    tools_map = {tool.__name__:tool for tool in tools}
    # This finds "SendMessageToUser":
    tool = tools_map.get(json_data["function"])
    if not tool:
        print(f"Error: unknown tool {json_data['function']}")
        return 1
    tool(**json_data["function_parameters"]).run()
```

```
    return 0


# Enum for the calculator tool.
class MathOperation(Enum):
    ADD = "add"
    SUBTRACT = "subtract"
    MULTIPLY = "multiply"
    DIVIDE = "divide"


# Simple pydantic calculator tool for the agent that can add, subtract,
# multiply, and divide. Docstring and description of fields will be used in
# system prompt.
class Calculator(BaseModel):
    """Perform a math operation on two numbers."""
    number_one: Union[int, float] = Field(..., description="First number.")
    operation: MathOperation = Field(..., description="Math operation to perform.")
    number_two: Union[int, float] = Field(..., description="Second number.")

    def run(self):
        if self.operation == MathOperation.ADD:
            return self.number_one + self.number_two
        elif self.operation == MathOperation.SUBTRACT:
            return self.number_one - self.number_two
        elif self.operation == MathOperation.MULTIPLY:
            return self.number_one * self.number_two
        elif self.operation == MathOperation.DIVIDE:
            return self.number_one / self.number_two
        else:
            raise ValueError("Unknown operation.")


def example_calculator(host):
    """Have the LLM ask to get a calculation done.

    Here the grammar gets generated by passing the available function models to
    generate_gbnf_grammar_and_documentation function. This also generates a
    documentation usable by the LLM.

    pydantic_model_list is the list of pydantic models outer_object_name is an
    optional name for an outer object around the actual model object. Like a
    "function" object with "function_parameters" which contains the actual model
    object. If None, no outer object will be generated outer_object_content is
    the name of outer object content.

    model_prefix is the optional prefix for models in the documentation. (Default="Output Model")
    fields_prefix is the prefix for the model fields in the documentation. (Default="Output Fields")
    """
    print("- example_calculator")
    tools = [SendMessageToUser, Calculator]
    gbnf_grammar, documentation = generate_gbnf_grammar_and_documentation(
        pydantic_model_list=tools, outer_object_name="function",
        outer_object_content="function_parameters", model_prefix="Function", fields_prefix="Parameters")
```

```
    system_message = "You are an advanced AI, tasked to assist the user by calling functions in JSON format. The following are the available functions and their parameters and types:\n\n" + documentation
    user_message1 = "What is 42 * 42?"
    prompt = f"<|im_start|>system\n{system_message}<|im_end|>\n<|im_start|>user\n{user_message1}<|im_end|>\n<|im_start|>assistant"
    text = create_completion(host, prompt, gbnf_grammar)
    json_data = json.loads(text)
    expected = {
        "function": "Calculator",
        "function_parameters": {
            "number_one": 42,
            "operation": "multiply",
            "number_two": 42
        }
    }
    if json_data != expected:
        print("  Result is not as expected!")
    tools_map = {tool.__name__:tool for tool in tools}
    # This finds "Calculator":
    tool = tools_map.get(json_data["function"])
    if not tool:
        print(f"Error: unknown tool {json_data['function']}")
        return 1
    result = tool(**json_data["function_parameters"]).run()
    print(f"  Call {json_data['function']} gave result {result}")
    return 0


class Category(Enum):
    """The category of the book."""
    Fiction = "Fiction"
    NonFiction = "Non-Fiction"


class Book(BaseModel):
    """Represents an entry about a book."""
    title: str = Field(..., description="Title of the book.")
    author: str = Field(..., description="Author of the book.")
    published_year: Optional[int] = Field(..., description="Publishing year of the book.")
    keywords: list[str] = Field(..., description="A list of keywords.")
    category: Category = Field(..., description="Category of the book.")
    summary: str = Field(..., description="Summary of the book.")


def example_struct(host):
    """A example structured output based on pydantic models.

    The LLM will create an entry for a Book database out of an unstructured
    text. We need no additional parameters other than our list of pydantic
    models.
    """
    print("- example_struct")
    tools = [Book]
```

```
    gbnf_grammar, documentation = generate_gbnf_grammar_and_documentation(pydantic_model_list=tools)
      system_message = "You are an advanced AI, tasked to create a dataset entry in JSON for a Book. The following is the expected output model:\n\n" + documentation
    text = """The Feynman Lectures on Physics is a physics textbook based on some lectures by Richard Feynman, a Nobel laureate who has sometimes been called "The Great Explainer". The lectures were presented before undergraduate students at the California Institute of Technology (Caltech), during 1961?1963. The book's co-authors are Feynman, Robert B. Leighton, and Matthew Sands."""
                                                                                prompt                    =
f"<|im_start|>system\n{system_message}<|im_end|>\n<|im_start|>user\n{text}<|im_end|>\n<|im_start|>assistant"
    text = create_completion(host, prompt, gbnf_grammar)
    json_data = json.loads(text)
    # In this case, there's no function nor function_parameters.
    # Here the result will vary based on the LLM used.
    keys = sorted(["title", "author", "published_year", "keywords", "category", "summary"])
    if keys != sorted(json_data.keys()):
        print(f"Unexpected result: {sorted(json_data.keys())}")
        return 1
    book = Book(**json_data)
    print(f"  As a Book object: %s" % book)
    return 0


def get_current_datetime(output_format: Optional[str] = None):
    """Get the current date and time in the given format.

    Args:
         output_format: formatting string for the date and time, defaults to '%Y-%m-%d %H:%M:%S'
    """
    return datetime.datetime.now().strftime(output_format or "%Y-%m-%d %H:%M:%S")


# Example function to get the weather.
def get_current_weather(location, unit):
    """Get the current weather in a given location"""
    if "London" in location:
        return json.dumps({"location": "London", "temperature": "42", "unit": unit.value})
    elif "New York" in location:
        return json.dumps({"location": "New York", "temperature": "24", "unit": unit.value})
    elif "North Pole" in location:
        return json.dumps({"location": "North Pole", "temperature": "-42", "unit": unit.value})
    return json.dumps({"location": location, "temperature": "unknown"})


def example_concurrent(host):
    """An example for parallel function calling with a Python function, a pydantic
    function model and an OpenAI like function definition.
    """
    print("- example_concurrent")
    # Function definition in OpenAI style.
    current_weather_tool = {
        "type": "function",
        "function": {
            "name": "get_current_weather",
            "description": "Get the current weather in a given location",
```

```
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "The city and state, e.g. San Francisco, CA",
                    },
                    "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
                },
                "required": ["location"],
            },
        },
    }
    # Convert OpenAI function definition into pydantic model.
    current_weather_tool_model = convert_dictionary_to_pydantic_model(current_weather_tool)
    # Add the actual function to a pydantic model.
                current_weather_tool_model    =    add_run_method_to_dynamic_model(current_weather_tool_model,
get_current_weather)

    # Convert normal Python function to a pydantic model.
    current_datetime_model = create_dynamic_model_from_function(get_current_datetime)

    tools = [SendMessageToUser, Calculator, current_datetime_model, current_weather_tool_model]
    gbnf_grammar, documentation = generate_gbnf_grammar_and_documentation(
        pydantic_model_list=tools, outer_object_name="function",
                        outer_object_content="params",   model_prefix="Function",   fields_prefix="Parameters",
list_of_outputs=True)
     system_message = "You are an advanced AI assistant. You are interacting with the user and with your
environment by calling functions. You call functions by writing JSON objects, which represent specific function
calls.\nBelow is a list of your available function calls:\n\n" + documentation
     text = """Get the date and time, get the current weather in celsius in London and solve the following
calculation: 42 * 42"""
                                                                                prompt                    =
f"<|im_start|>system\n{system_message}<|im_end|>\n<|im_start|>user\n{text}<|im_end|>\n<|im_start|>assistant"
    text = create_completion(host, prompt, gbnf_grammar)
    json_data = json.loads(text)
    expected = [
      {
        "function": "get_current_datetime",
        "params": {
          "output_format": "%Y-%m-%d %H:%M:%S"
        }
      },
      {
        "function": "get_current_weather",
        "params": {
          "location": "London",
          "unit": "celsius"
        }
      },
      {
        "function": "Calculator",
        "params": {
          "number_one": 42,
```