# KV-Cache Simulator — Code documentation

This document contains the main source files and explanations for the KV-cache simulator. Each file is followed by a short description of the purpose and a per-function summary where applicable.

## main.py

Top-level runner: loads config, generates trace, runs simulation loop, logs results and writes plots.

```
"""
Main entry point for the KV-cache simulator.
"""
import yaml
from core.engine import SimulatorEngine
from memory_models.monolithic_kv import MonolithicKV
from memory_models.paged_kv import PagedKV
from memory_models.paged_compressed_kv import PagedCompressedKV
from results.logger import Logger
from results.stats import compute_stats
from results.plotter import plot_records
from utils.helpers import generate_synthetic_trace
import sys


def load_config(path):
    with open(path, 'r') as f:
        return yaml.safe_load(f)


def main(config_path):
    config = load_config(config_path)
    logger = Logger()
    trace = generate_synthetic_trace(config['simulation_steps'], 'mixed')

    # Baseline: Monolithic KV
    monolithic = MonolithicKV(config['monolithic_kv_size'])
    # Paged KV
    paged = PagedKV(config['paged_kv_num_pages'], config['paged_kv_page_size'])
    # Paged + Compression Gate
    paged_compressed = PagedCompressedKV(
        config['paged_kv_num_pages'],
        config['paged_kv_page_size'],
        config['compression_ratio'],
        config['pressure_threshold']
    )

    # Track allocations by id so we can free them later per model
    alloc_table = {}  # alloc_id -> size
    allocs_monolithic = {}  # alloc_id -> amount
    allocs_paged = {}  # alloc_id -> num_blocks
    allocs_paged_compressed = {}  # alloc_id -> num_blocks

    # Run simulation for each model and record per-step state for all three
    for step, event in enumerate(trace):
        if event['op'] == 'alloc':
            alloc_id = event['id']
            size = event['size']

            # Monolithic allocate
            ok_m = monolithic.allocate(size)
            if ok_m:
                allocs_monolithic[alloc_id] = size

            # Paged allocations use ceil conversion
            page_size = config['paged_kv_page_size']
```

```
            blocks_needed = (size + page_size - 1) // page_size
            ok_p = paged.allocate(blocks_needed)
            if ok_p:
                allocs_paged[alloc_id] = blocks_needed

            ok_pc = paged_compressed.allocate(blocks_needed)
            if ok_pc:
                allocs_paged_compressed[alloc_id] = blocks_needed
                # mark the newly allocated pages as accessed at this step for LRU
                # find indices of pages allocated (state==1) that have last_access==0
                allocated_indices = [i for i, s in enumerate(paged_compressed.pages) if s == 1 and paged
                paged_compressed.touch_pages(allocated_indices, step)

            alloc_table[alloc_id] = size

        elif event['op'] == 'free':
            alloc_id = event['id']
            # free for monolithic
            if alloc_id in allocs_monolithic:
                monolithic.free(allocs_monolithic.pop(alloc_id))
            # free for paged
            if alloc_id in allocs_paged:
                paged.free(allocs_paged.pop(alloc_id))
            # free for paged_compressed
            if alloc_id in allocs_paged_compressed:
                paged_compressed.free(allocs_paged_compressed.pop(alloc_id))

        # Compute memory usage for paged models in bytes (or same units as req)
        page_size = config['paged_kv_page_size']
        paged_used_pages = sum(1 for x in paged.pages if x == 1)
        mem_paged = paged_used_pages * page_size

        pc_used_pages = sum(1 for x in paged_compressed.pages if x == 1)
        pc_compressed_pages = sum(1 for x in paged_compressed.pages if x == 2)
        mem_paged_compressed = pc_used_pages * page_size + pc_compressed_pages * page_size * config['com

        # Fragmentation: fraction of free pages (for paged models), monolithic placeholder 0
        frag_paged = (paged.num_pages - paged_used_pages) / paged.num_pages
        frag_paged_compressed = (paged_compressed.num_pages - (pc_used_pages + pc_compressed_pages)) / p
        frag_monolithic = 0.0

        throughput_val = event['size'] if event['op'] == 'alloc' else 0

        logger.log({
            'step': step,
            'event': event,
            'throughput': throughput_val,
            'memory_monolithic': monolithic.usage,
            'memory_paged': mem_paged,
            'memory_paged_compressed': mem_paged_compressed,
            'fragmentation_monolithic': frag_monolithic,
            'fragmentation_paged': frag_paged,
            'fragmentation_paged_compressed': frag_paged_compressed,
        })

    # Compute aggregated statistics and produce comparison plots
    stats = compute_stats(logger.records)
    print('Simulation stats:', stats)
    logger.save('results.txt')

    # Create comparison plots (saved to results/)
    plot_records(logger.records, out_dir='results')

if __name__ == '__main__':
    config_path = sys.argv[1] if len(sys.argv) > 1 else 'config/default_config.yaml'
    main(config_path)
```

## *Function summaries:*

**load_config(path)**: Load YAML config file and return dict.

**main(config_path)**: Main entry: loads config, instantiates models, processes trace events, logs state, computes stats, and plots results.

## utils/helpers.py

Synthetic trace generator. Produces alloc/free events with configurable lifetimes.

```
"""
Helper functions for the simulator.
"""
import random


def generate_synthetic_trace(num_steps, workload_type, free_probability=0.3, lifetime_range=(5, 50)):
    """
    Generate a sequence of events with allocations and frees.

    Each event is a dict:
      - {'op': 'alloc', 'id': int, 'size': int}
      - {'op': 'free', 'id': int}

    We schedule frees for allocations after a random lifetime (within lifetime_range).
    free_probability is the fraction of allocated objects that are eligible to be freed before end.
    """
    trace = [None] * num_steps
    alloc_id = 0
    scheduled_frees = {}

    for t in range(num_steps):
        # Inject any scheduled frees for this time
        if t in scheduled_frees:
            # schedule frees at this time (may be multiple)
            for a_id in scheduled_frees[t]:
                trace[t] = {'op': 'free', 'id': a_id}
                # Only one event per step in this simple generator; if multiple scheduled, keep the last

        # If there's already a free scheduled at this step, sometimes also emit an alloc
        if trace[t] is None or random.random() < 0.5:
            # create an allocation event
            if workload_type == 'short':
                size = random.randint(1, 4)
            elif workload_type == 'long':
                size = random.randint(8, 32)
            else:
                size = random.randint(1, 32)

            trace[t] = {'op': 'alloc', 'id': alloc_id, 'size': size}

            # schedule a free for this allocation with some probability
            if random.random() < free_probability:
                lifetime = random.randint(lifetime_range[0], lifetime_range[1])
                free_time = min(num_steps - 1, t + lifetime)
                scheduled_frees.setdefault(free_time, []).append(alloc_id)

            alloc_id += 1

    # Any remaining scheduled frees that didn't get placed overwrite some allocs near the end
    for t, ids in scheduled_frees.items():
        for a_id in ids:
            if t < num_steps:
                trace[t] = {'op': 'free', 'id': a_id}

    return trace
```

### *Function summaries:*

**generate_synthetic_trace(num_steps, workload_type, free_probability, lifetime_range)**: Generates a sequence of allocation/free events; allocations are given IDs and frees scheduled after a random lifetime.

## memory_models/monolithic_kv.py

Monolithic allocator model: single scalar usage, allocate/free.

```
"""
Monolithic KV-cache model (baseline).
"""

class MonolithicKV:
    def __init__(self, size):
        self.size = size
        self.usage = 0

    def allocate(self, amount):
        if self.usage + amount <= self.size:
            self.usage += amount
            return True
        return False

    def free(self, amount):
        self.usage = max(0, self.usage - amount)
```

### *Function summaries:*

**MonolithicKV.__init__(size)**: Create monolithic allocator with capacity `size`.

**MonolithicKV.allocate(amount)**: Attempt to allocate `amount`; increments usage if enough capacity and returns True/False.

**MonolithicKV.free(amount)**: Frees `amount` by decreasing usage, not below zero.

## memory_models/paged_kv.py

Paged allocator: fixed pages, allocate/free by blocks.

```
"""
Paged KV-cache model.
"""

class PagedKV:
    def __init__(self, num_pages, page_size):
        self.num_pages = num_pages
        self.page_size = page_size
        self.pages = [0] * num_pages  # 0: free, 1: used

    def allocate(self, num_blocks):
        allocated = 0
        for i in range(self.num_pages):
            if self.pages[i] == 0:
                self.pages[i] = 1
                allocated += 1
                if allocated == num_blocks:
                    return True
        return False

    def free(self, num_blocks):
        freed = 0
```

```
        for i in range(self.num_pages):
            if self.pages[i] == 1:
                self.pages[i] = 0
                freed += 1
                if freed == num_blocks:
                    return True
        return False
```

### *Function summaries:*

**PagedKV.__init__(num_pages, page_size)**: Initialize pages and sizes.

**PagedKV.allocate(num_blocks)**: Find free pages and mark them used for `num_blocks`; return True if allocated.

**PagedKV.free(num_blocks)**: Free up to `num_blocks` used pages.

## memory_models/paged_compressed_kv.py

Paged allocator with compression gate and LRU-informed compression.

```
"""
Paged KV-cache with compression gate.
"""

class PagedCompressedKV:
    def __init__(self, num_pages, page_size, compression_ratio, pressure_threshold):
        self.num_pages = num_pages
        self.page_size = page_size
        self.compression_ratio = float(compression_ratio)
        self.pressure_threshold = pressure_threshold
        self.pages = [0] * num_pages  # 0: free, 1: used, 2: compressed
        # LRU: per-page last access timestamp (higher = more recent). 0 = never used.
        self.last_access = [0] * num_pages


    def effective_usage_pages(self):
        """Return effective usage in page-equivalents: used pages count as 1, compressed count as compre
        used = sum(1 for s in self.pages if s == 1)
        compressed = sum(1 for s in self.pages if s == 2)
        return used + compressed * self.compression_ratio

    def allocate(self, num_blocks):
        # Check whether there's enough effective capacity
        if self.effective_usage_pages() + num_blocks > self.num_pages:
            # try to compress more to make room
            self.compress_cold_blocks(target_free=num_blocks)
            if self.effective_usage_pages() + num_blocks > self.num_pages:
                return False

        # Ensure there are enough physical free pages. If not, attempt to compress to free physical page
        free_pages = [i for i, s in enumerate(self.pages) if s == 0]
        if len(free_pages) < num_blocks:
            # attempt compression to free physical slots
            self.compress_cold_blocks(target_free=num_blocks - len(free_pages))
            free_pages = [i for i, s in enumerate(self.pages) if s == 0]
            if len(free_pages) < num_blocks:
                return False

        # Allocate into free pages
        for i in free_pages[:num_blocks]:
            self.pages[i] = 1
            self.last_access[i] = 0  # will be set by caller via touch
        # After allocation, check compression gate
        self.check_compression_gate()
        return True
```

```
def free(self, num_blocks):
    # Free used pages first, then compressed pages
    freed = 0
    for i in range(self.num_pages):
        if self.pages[i] == 1:
            self.pages[i] = 0
            self.last_access[i] = 0
            freed += 1
            if freed == num_blocks:
                return True
    for i in range(self.num_pages):
        if self.pages[i] == 2:
            self.pages[i] = 0
            self.last_access[i] = 0
            freed += 1
            if freed == num_blocks:
                return True
    return False

def check_compression_gate(self):
    pressure = self.effective_usage_pages() / self.num_pages
    if pressure > self.pressure_threshold:
        # compress to try to bring pressure down
        self.compress_cold_blocks()

def touch_pages(self, indices, timestamp):
    """Mark given page indices as accessed at timestamp (for LRU)."""
    for i in indices:
        if 0 <= i < self.num_pages:
            self.last_access[i] = timestamp

def compress_cold_blocks(self, target_free=0):
    """
    Compress cold/used pages to free up capacity.

    Strategy (heuristic): group several used pages into a single compressed page based on compressio
    For a compression_ratio r, group_size = int(round(1 / r)) pages can be packed into 1 compressed
    This will free group_size - 1 physical pages per group.

    If target_free > 0, we'll try to free at least that many physical pages by performing groups.
    """
    if self.compression_ratio <= 0 or self.compression_ratio >= 1:
        # no effective compression possible
        return

    group_size = max(2, int(round(1.0 / self.compression_ratio)))

    # indices of currently used pages
    used_indices = [i for i, s in enumerate(self.pages) if s == 1]
    if not used_indices:
        return

    freed_total = 0
    # For better results, sort used pages by last_access ascending (cold first)
    used_indices.sort(key=lambda x: self.last_access[x])

    # iterate in groups of group_size over cold pages first
    i = 0
    while i + group_size <= len(used_indices):
        group = used_indices[i:i+group_size]
        # compress the group into one compressed page: pick the coldest page as the compressed targe
        group_sorted = sorted(group, key=lambda x: self.last_access[x])
        first = group_sorted[0]
        others = group_sorted[1:]
        self.pages[first] = 2
        # reset last_access for compressed page to recent (simulate compression activity)
        self.last_access[first] = max(self.last_access) + 1 if self.last_access else 1
        for idx in others:
            self.pages[idx] = 0
            self.last_access[idx] = 0
        freed_total += (group_size - 1)
        i += group_size
```

```
    if target_free and freed_total >= target_free:
        break

# try compressing smaller leftover group into 1 compressed page
if target_free and freed_total < target_free:
    # take remaining used pages if any
    remaining = [i for i, s in enumerate(self.pages) if s == 1]
    if remaining:
        # compress all remaining into one compressed page
        first = remaining[0]
        others = remaining[1:]
        self.pages[first] = 2
        for idx in others:
            self.pages[idx] = 0
        freed_total += len(others)
```

### *Function summaries:*

**PagedCompressedKV.__init__**: Initialize pages, compression_ratio, pressure_threshold, and LRU timestamps.

**effective_usage_pages()**: Return effective pages used accounting for compressed pages.

**allocate(num_blocks)**: Attempt allocation: may compress cold pages to make room; returns True/False.

**free(num_blocks)**: Free used pages then compressed pages.

**compress_cold_blocks(target_free=0)**: Perform LRU-guided grouping compression to free physical pages.

**touch_pages(indices, timestamp)**: Update LRU timestamps for pages when accessed.

## results/logger.py

Logger: collects per-step records and saves text + CSV output.

```
"""
Logger for simulation results.
"""

class Logger:
    def __init__(self):
        self.records = []

    def log(self, data):
        self.records.append(data)

    def save(self, filename):
        # Save human-readable text log
        with open(filename, 'w') as f:
            for record in self.records:
                f.write(f"{record}\n")

        # Save CSV for easier analysis
        csv_name = filename.replace('.txt', '.csv')
        import csv
        if not self.records:
            return
        # determine headers from first record
        headers = []
        # flatten event keys
        first = self.records[0]
        for k in ['step', 'event', 'throughput', 'memory_monolithic', 'memory_paged', 'memory_paged_comp
            if k in first:
                headers.append(k)
```

```
        with open(csv_name, 'w', newline='') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=headers)
            writer.writeheader()
            for r in self.records:
                # convert event dict to string for CSV
                row = {k: r.get(k, '') for k in headers}
                if 'event' in row and isinstance(row['event'], dict):
                    row['event'] = str(row['event'])
                writer.writerow(row)
```

### *Function summaries:*

**Logger.log(data)**: Append a record to in-memory list.

**Logger.save(filename)**: Write text log and CSV derived from records.

## results/plotter.py

Plotter: creates PNG comparison plots for memory, fragmentation, and throughput.

```
"""Plotting utilities for simulation results.
Saves one image per comparison metric into an output directory.
"""
import os
import matplotlib.pyplot as plt


def plot_records(records, out_dir='results'):
    os.makedirs(out_dir, exist_ok=True)

    steps = [r['step'] for r in records]

    # Memory over time for each model
    mem_mono = [r['memory_monolithic'] for r in records]
    mem_paged = [r['memory_paged'] for r in records]
    mem_pc = [r['memory_paged_compressed'] for r in records]

    plt.figure(figsize=(10, 6))
    plt.plot(steps, mem_mono, label='Monolithic')
    plt.plot(steps, mem_paged, label='Paged')
    plt.plot(steps, mem_pc, label='Paged+Compressed')
    plt.xlabel('Step')
    plt.ylabel('Memory usage (units)')
    plt.title('Memory usage over time')
    plt.legend()
    plt.grid(True)
    mem_path = os.path.join(out_dir, 'memory_usage_comparison.png')
    plt.savefig(mem_path)
    plt.close()

    # Fragmentation comparison
    frag_mono = [r['fragmentation_monolithic'] for r in records]
    frag_paged = [r['fragmentation_paged'] for r in records]
    frag_pc = [r['fragmentation_paged_compressed'] for r in records]

    plt.figure(figsize=(10, 6))
    plt.plot(steps, frag_mono, label='Monolithic')
    plt.plot(steps, frag_paged, label='Paged')
    plt.plot(steps, frag_pc, label='Paged+Compressed')
    plt.xlabel('Step')
    plt.ylabel('Fragmentation (fraction)')
    plt.title('Fragmentation over time')
    plt.legend()
    plt.grid(True)
    frag_path = os.path.join(out_dir, 'fragmentation_comparison.png')
    plt.savefig(frag_path)
    plt.close()
```

```
    # Throughput over time
    throughput = [r['throughput'] for r in records]
    plt.figure(figsize=(10, 6))
    plt.plot(steps, throughput, label='Throughput')
    plt.xlabel('Step')
    plt.ylabel('Request size')
    plt.title('Request sizes (throughput) over time')
    plt.grid(True)
    thr_path = os.path.join(out_dir, 'throughput.png')
    plt.savefig(thr_path)
    plt.close()

    print(f"Saved plots: {mem_path}, {frag_path}, {thr_path}")
```

### *Function summaries:*

**plot_records(records, out_dir)**: Render three PNGs for memory, fragmentation, throughput and save to out_dir.

## results/stats.py

Aggregate statistics computation for each model.

```
"""
Statistics computation for simulation results.
"""

def compute_stats(records):
    if not records:
        return {}

    def peak_avg(values):
        return {'peak': max(values), 'avg': sum(values) / len(values)}

    mem_mono = [r['memory_monolithic'] for r in records]
    mem_paged = [r['memory_paged'] for r in records]
    mem_pc = [r['memory_paged_compressed'] for r in records]

    frag_mono = [r['fragmentation_monolithic'] for r in records]
    frag_paged = [r['fragmentation_paged'] for r in records]
    frag_pc = [r['fragmentation_paged_compressed'] for r in records]

    throughput = [r['throughput'] for r in records]

    return {
        'monolithic': peak_avg(mem_mono),
        'paged': peak_avg(mem_paged),
        'paged_compressed': peak_avg(mem_pc),
        'fragmentation': {
            'monolithic': sum(frag_mono) / len(frag_mono),
            'paged': sum(frag_paged) / len(frag_paged),
            'paged_compressed': sum(frag_pc) / len(frag_pc),
        },
        'throughput_avg': sum(throughput) / len(throughput),
    }
```

### *Function summaries:*

**compute_stats(records)**: Calculate peak/avg memory per model, average fragmentation and throughput.

## examples/demo_payload.py

Small demo script that generates a short trace and prints allocation results.

```
import sys
from pathlib import Path

# Ensure the project root is on sys.path so local imports work when running this script
PROJECT_ROOT = Path(__file__).resolve().parents[1]
sys.path.insert(0, str(PROJECT_ROOT))

from utils.helpers import generate_synthetic_trace
from memory_models.monolithic_kv import MonolithicKV
from memory_models.paged_kv import PagedKV
from memory_models.paged_compressed_kv import PagedCompressedKV

CONFIG = {
    'simulation_steps': 10,
    'monolithic_kv_size': 100,
    'paged_kv_num_pages': 10,
    'paged_kv_page_size': 10,
    'compression_ratio': 0.5,
    'pressure_threshold': 0.8,
}

trace = generate_synthetic_trace(CONFIG['simulation_steps'], 'mixed')
print('Generated trace:', trace)

m = MonolithicKV(CONFIG['monolithic_kv_size'])
p = PagedKV(CONFIG['paged_kv_num_pages'], CONFIG['paged_kv_page_size'])
pc = PagedCompressedKV(
    CONFIG['paged_kv_num_pages'], CONFIG['paged_kv_page_size'],
    CONFIG['compression_ratio'], CONFIG['pressure_threshold']
)

print('\nSimulating allocations:')
for i, req in enumerate(trace):
    m_ok = m.allocate(req)
    p_ok = p.allocate(req // CONFIG['paged_kv_page_size'] + 1)
    pc_ok = pc.allocate(req // CONFIG['paged_kv_page_size'] + 1)
    print(f"Step {i}: req={req:2d} | Monolithic: ok={m_ok:5} usage={m.usage:3d} | Paged: ok={p_ok:5} use
```

### *Function summaries:*

No additional function summaries available.

## config/default_config.yaml

Default simulation configuration values.

```
# Default configuration for the KV-cache simulator
simulation_steps: 10000
monolithic_kv_size: 4096
paged_kv_num_pages: 128
paged_kv_page_size: 32
compression_ratio: 0.5
pressure_threshold: 0.8
```

### *Function summaries:*

No additional function summaries available.

# README.md

Project README (also present in repo).

````
# KV-Cache Simulator — Memory Design & Implementation

This repository contains a small simulator to experiment with different in-memory key-value cache layout

## Project structure

```
simulator/
■■■ main.py                    # Main entry point to run the simulation
■■■ README.md                  # This file
■■■ config/
■   ■■■ default_config.yaml    # Default simulation parameters
■■■ core/
■   ■■■ engine.py              # Simulation engine scaffold (event loop placeholder)
■■■ examples/
■   ■■■ demo_payload.py        # Small demo script that prints a sample trace and allocations
■■■ interface/
■   ■■■ cli.py                 # Simple CLI parser (placeholder)
■■■ memory_models/
■   ■■■ monolithic_kv.py       # Monolithic (single block) allocator model
■   ■■■ paged_kv.py            # Simple fixed-page allocator model
■   ■■■ paged_compressed_kv.py# Paged allocator with compression gate (improved)
■■■ results/
■   ■■■ logger.py              # Simple logger that records step-by-step state
■   ■■■ plotter.py             # Plotting helper that saves PNG comparisons
■   ■■■ stats.py               # Aggregated stats calculator
■■■ utils/
■   ■■■ helpers.py             # Trace generator and small utilities
■■■ examples/                  # Example scripts
■■■ results.txt                # Output log produced by the simulation
```

## How payloads are generated

Payloads in this simulator are synthetic events (allocations and frees). The generator is
`utils/helpers.py::generate_synthetic_trace(num_steps, workload_type, free_probability, lifetime_range)`

Event format:
- Allocation: `{'op': 'alloc', 'id': <int>, 'size': <int>}`
- Free: `{'op': 'free', 'id': <int>}`

Behavior:
- The generator schedules frees for previously allocated IDs after a randomly chosen lifetime (within `l
- `free_probability` controls whether a given allocation will be scheduled to be freed at all (so you ca

Workload types still control the size distribution for allocation events:
- `short`: small allocations (random 1..4)
- `long`: larger allocations (random 8..32)
- `mixed`: random 1..32 (default used by `main.py`)

Each allocation's `size` is treated as a request size (in abstract units). For the paged models this siz

## Memory model details

### MonolithicKV (file: `memory_models/monolithic_kv.py`)

- Concept: single contiguous pool of memory tracked by `usage` and `size`.
- API: `allocate(amount)` and `free(amount)`
  - `allocate(amount)` succeeds if `usage + amount <= size`, then increments `usage`.
  - `free(amount)` reduces `usage` but never below zero.
- Behavior: simple and fast; no fragmentation tracking, no paging or compression.
- Use cases: baseline for measuring raw memory consumption and peak usage.

### PagedKV (file: `memory_models/paged_kv.py`)

- Concept: divide the pool into `num_pages` pages, each of `page_size` units.
- State: `pages` list stores state per page: `0` = free, `1` = used.
````

- Allocation: requests are converted to `num_blocks = ceil(request / page_size)` (main uses the correct
- Fragmentation: can be approximated by counting free vs used pages.
- Limitations: greedy allocation (first free pages), no reuse pattern beyond simple `free`.

### PagedCompressedKV (file: `memory_models/paged_compressed_kv.py`)

- Concept: similar to `PagedKV` but supports a compression gate which packs several used pages into fewe
- States per page: `0` = free, `1` = used (uncompressed), `2` = compressed.
- Compression model (implemented):
  - `compression_ratio` is a fraction in (0,1). A compressed page counts as `compression_ratio` page-equ
  - The model uses an LRU-informed compression policy: each page records a `last_access` timestamp. When
  - A heuristic groups `group_size = round(1 / compression_ratio)` used pages and packs them into a sing
  - The allocator checks `effective_usage_pages = used + compressed * compression_ratio` and attempts to
- Freeing: `free` prefers to free uncompressed used pages and then compressed pages.
- Notes: This is still a heuristic model but now respects recency via LRU timestamps and preferentially

## Configuration

Default config is in `config/default_config.yaml`. Typical keys:
- `simulation_steps`: number of steps to simulate
- `monolithic_kv_size`: capacity of monolithic model (units)
- `paged_kv_num_pages`: number of pages used by page-based models
- `paged_kv_page_size`: page size in units
- `compression_ratio`: compression savings for compressed page (0 < r < 1)
- `pressure_threshold`: fraction of usage above which compression is triggered

## Running the simulator

Install dependencies (recommended in a virtualenv):

```bash
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Then run the simulation:

```bash
python3 main.py              # uses config/default_config.yaml by default
python3 main.py config/default_config.yaml
```

Outputs:
- `results.txt` (text log of per-step records)
- `results.csv` (CSV with one row per step: step,event,throughput,memory_monolithic,memory_paged,memory_
- Plots saved in `results/`:
  - `memory_usage_comparison.png`
  - `fragmentation_comparison.png`
  - `throughput.png`

## Files of interest

- `main.py`: glue code — loads config, generates trace, instantiates models, logs state each step, compu
- `utils/helpers.py`: the synthetic trace generator.
- `memory_models/*`: three memory models.
- `results/plotter.py`: plotting helper using matplotlib to generate PNGs.
- `results/stats.py`: computes aggregated per-model stats (peak and average).

## Example workflow and experiments

- Compare different compression ratios: change `compression_ratio` in `config/default_config.yaml` and r
-- Add deallocation/lifetimes: the project already includes event-based traces (alloc/free). You can tun
-- LRU-based compression is implemented in `memory_models/paged_compressed_kv.py`. Try varying `compress
-- Export CSV: the logger now writes `results.csv` alongside `results.txt` for easy analysis.

## Notes & limitations

- The simulator is intentionally small and illustrative. The PagedCompressedKV compression is a heuristi
- The engine (`core/engine.py`) is a scaffold; it does not currently process events or simulate time bey

## Contact

```
If you want, I can:
- Add alloc/free events to the trace generator and update `main.py` to handle lifetimes.
- Implement LRU-based compression or asynchronous compression.
- Export CSV outputs for deeper analysis.

Pick the next enhancement you want and I will implement it.
```

### *Function summaries:*

No additional function summaries available.