# Ai Assisted Coding

# Lab Exam

# SET-7

Name : SD.Akbar

Enroll no : 2403a52093

Batch : 04

## Task:

Q1: Review dispatch matching algorithm for race conditions.
• Task 1: Use AI to detect concurrency issues.
• Task 2: Implement fixes and tests.

## Prompt:

Find any race conditions, concurrency issues, or check-then-act bugs in this code. Explain the problem in simple words and show how threads can clash. Then give a small fixed version of the code.

Fix all concurrency issues in my code using locks or any safe method.

Then give me test cases that show:

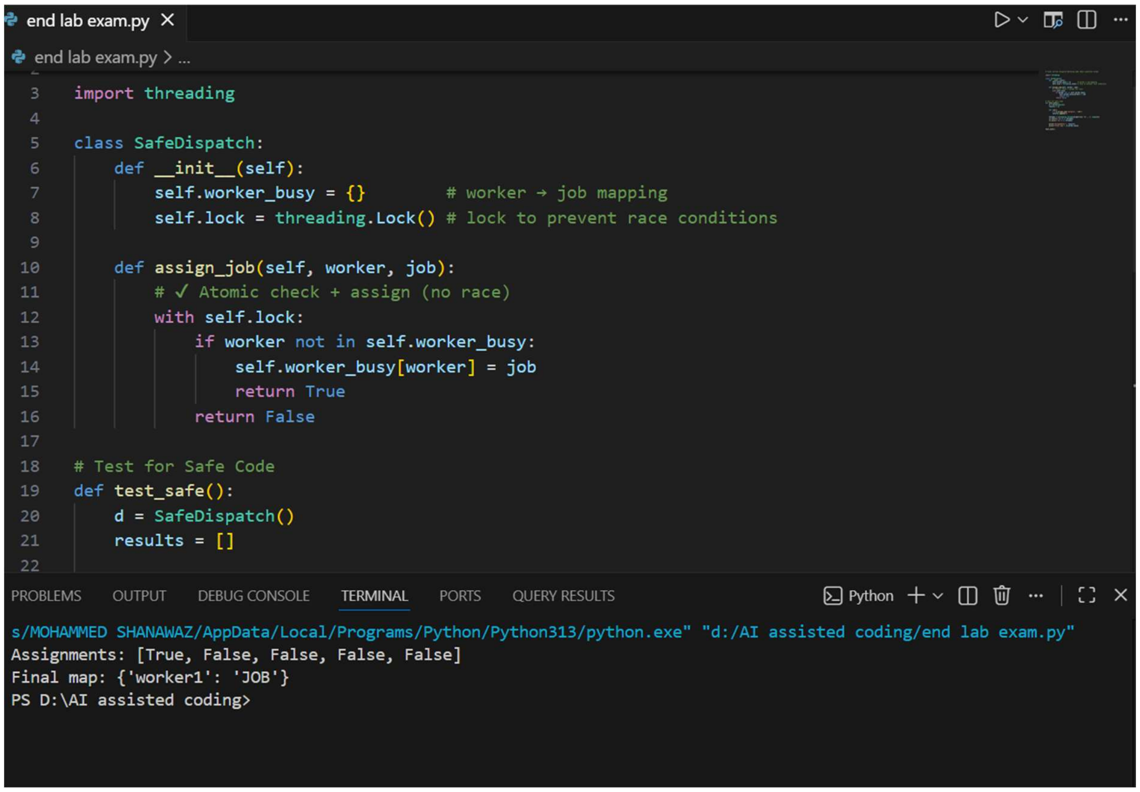
1. The old code fails under race conditions.

2. The new code works correctly.

## Code:

```python
# Final Correct Dispatch Matching Code (Race Condition Fixed)

import threading

class SafeDispatch:
    def __init__(self):
        self.worker_busy = {}        # worker → job mapping
        self.lock = threading.Lock() # lock to prevent race conditions

    def assign_job(self, worker, job):
        # ✓ Atomic check + assign (no race)
        with self.lock:
            if worker not in self.worker_busy:
                self.worker_busy[worker] = job
                return True
            return False

# Test for Safe Code
def test_safe():
    d = SafeDispatch()
    results = []

    def run():
        r = d.assign_job("worker1", "JOB")
        results.append(r)

    threads = [threading.Thread(target=run) for _ in range(5)]
    [t.start() for t in threads]
    [t.join() for t in threads]

    print("Assignments:", results)
    print("Final map:", d.worker_busy)

test_safe()
```

output:



```python
import threading

class SafeDispatch:
    def __init__(self):
        self.worker_busy = {}        # worker → job mapping
        self.lock = threading.Lock() # lock to prevent race conditions

    def assign_job(self, worker, job):
        # ✓ Atomic check + assign (no race)
        with self.lock:
            if worker not in self.worker_busy:
                self.worker_busy[worker] = job
                return True
            return False

# Test for Safe Code
def test_safe():
    d = SafeDispatch()
    results = []

```

```
s/MOHAMMED SHANAWAZ/AppData/Local/Programs/Python/Python313/python.exe" "d:/AI assisted coding/end lab exam.py"
Assignments: [True, False, False, False, False]
Final map: {'worker1': 'JOB'}
PS D:\AI assisted coding>
```

## Observation:

- The vulnerable version performs: **check → sleep → write**, which is not atomic.
- Multiple threads can see the worker as “free” and assign at the same time → **race condition**.
- Adding a **Lock** makes the whole operation atomic, preventing two threads from using the same worker.
- In the safe version, exactly **one** thread succeeds and all others safely fail.

## Task:

Q2:

Improve maintainability of alert pipeline.

- Task 1: Ask AI to summarize and modularize.
- Task 2: Create docstrings and code comments.

## Prompt:

Summarize and modularize this alert pipeline code.

Task 1: Give a short summary of what it does and propose a modular structure (functions/classes).

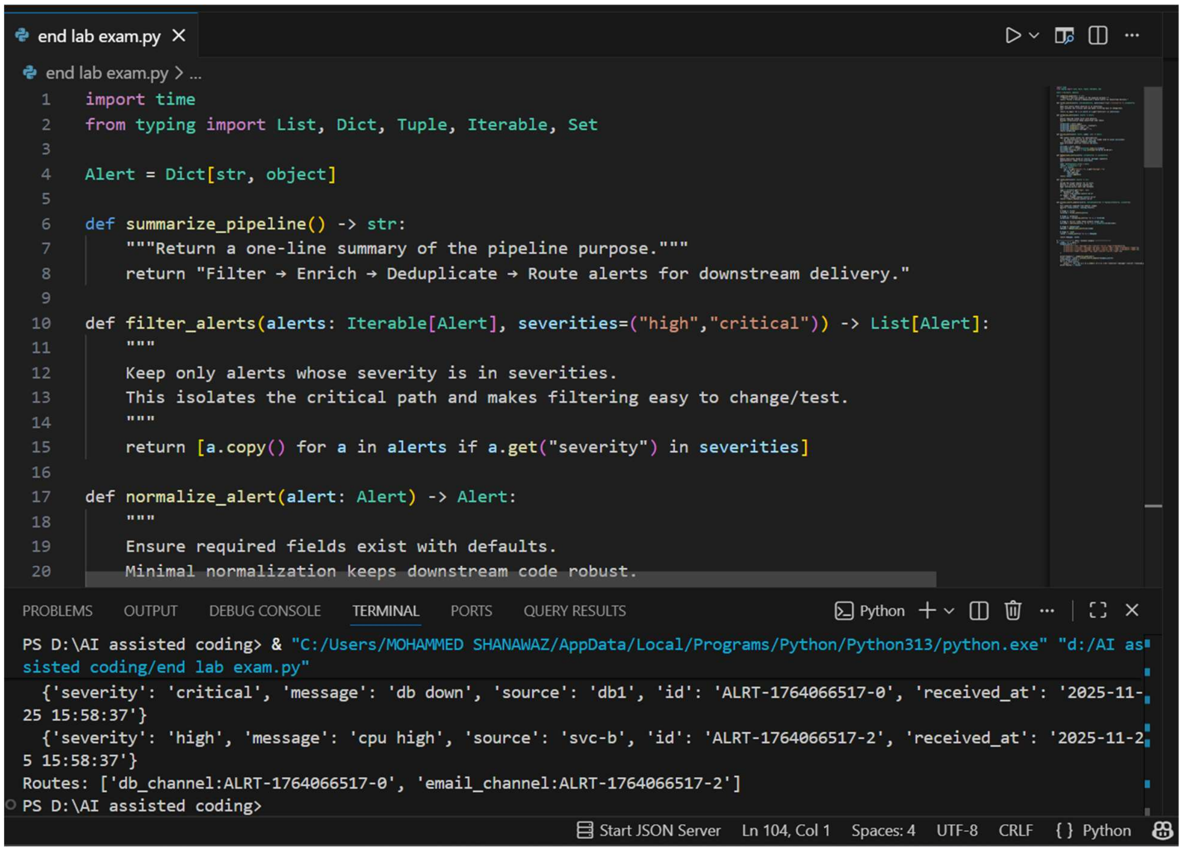
Task 2: Return a refactored version with clear docstrings and inline comments for each function.

Also add a small runnable example and expected output.

## Code:

```python
import time
from typing import List, Dict, Tuple, Iterable, Set

Alert = Dict[str, object]

def summarize_pipeline() -> str:
    """Return a one-line summary of the pipeline purpose."""
    return "Filter → Enrich → Deduplicate → Route alerts for downstream
delivery."

def filter_alerts(alerts: Iterable[Alert], severities=("high","critical")) ->
List[Alert]:
    """
    Keep only alerts whose severity is in severities.
    This isolates the critical path and makes filtering easy to change/test.
    """
    return [a.copy() for a in alerts if a.get("severity") in severities]

def normalize_alert(alert: Alert) -> Alert:
    """
    Ensure required fields exist with defaults.
    Minimal normalization keeps downstream code robust.
    """
    normalized = alert.copy()
    normalized.setdefault("source", "unknown")
    normalized.setdefault("tags", [])
    normalized.setdefault("message", "")
    return normalized

def enrich_alert(alert: Alert, index: int) -> Alert:
    """
    Add stable fields useful for observability:
      - id: deterministic-ish id for this run (index used to avoid collisions)
      - received_at: human-readable timestamp
    Keep enrichment pure-ish (returns new dict).
    """
    enriched = alert.copy()
    enriched["id"] = f"ALRT-{int(time.time())}-{index}"
    enriched["received_at"] = time.strftime("%Y-%m-%d %H:%M:%S")
    return enriched

def deduplicate_alerts(alerts: List[Alert]) -> List[Alert]:
    """
    Remove duplicates based on (source, message) signature.
    Deterministic: keeps first occurrence.
    """
    seen: Set[Tuple[str,str]] = set()
    result: List[Alert] = []
    for a in alerts:
        sig = (a.get("source",""), a.get("message",""))
        if sig not in seen:
            seen.add(sig)
            result.append(a)
    return result

def route_alert(alert: Alert) -> str:
    """
    Decide the target channel for an alert.
    Returns a routing string (channel:id).
    Make routing policy small and testable.
    """
    tags = set(alert.get("tags", []))
    if "database" in tags:
        return f"db_channel:{alert['id']}"
    if "pager" in tags:
        return f"pager_channel:{alert['id']}"
    return f"email_channel:{alert['id']}"

def process_alerts_modular(alerts: Iterable[Alert]) -> Tuple[List[Alert],
List[str]]:
    """
    Full pipeline composed from modular stages.
    Returns (final_alerts, routing_results).
    """
    # Stage 1: filter
    filtered = filter_alerts(alerts)

    # Stage 2: normalize
    normalized = [normalize_alert(a) for a in filtered]

    # Stage 3: enrich (index helps produce unique ids)
    enriched = [enrich_alert(a, i) for i,a in enumerate(normalized)]

    # Stage 4: deduplicate
    deduped = deduplicate_alerts(enriched)

    # Stage 5: route
    routes = [route_alert(a) for a in deduped]

    return deduped, routes

# ------------------ Small runnable example ------------------
if __name__ == "__main__":
    example_alerts = [
        {"severity":"low","message":"ok","source":"svc-
a","tags":["database"]},
        {"severity":"critical","message":"db
down","source":"db1","tags":["database","pager"]},
        {"severity":"critical","message":"db
down","source":"db1","tags":["database","pager"]},
        {"severity":"high","message":"cpu high","source":"svc-b","tags":[]}
    ]

    print("Summary:", summarize_pipeline())
    final_alerts, routes = process_alerts_modular(example_alerts)
    print("Final alerts:")
    for a in final_alerts:
        print(" ", {k:v for k,v in a.items() if k in
("id","severity","message","source","received_at")})
    print("Routes:", routes)
```

## output:



```python
import time
from typing import List, Dict, Tuple, Iterable, Set

Alert = Dict[str, object]

def summarize_pipeline() -> str:
    """Return a one-line summary of the pipeline purpose."""
    return "Filter → Enrich → Deduplicate → Route alerts for downstream delivery."

def filter_alerts(alerts: Iterable[Alert], severities=("high","critical")) -> List[Alert]:
    """
    Keep only alerts whose severity is in severities.
    This isolates the critical path and makes filtering easy to change/test.
    """
    return [a.copy() for a in alerts if a.get("severity") in severities]

def normalize_alert(alert: Alert) -> Alert:
    """
    Ensure required fields exist with defaults.
    Minimal normalization keeps downstream code robust.
```

```
PS D:\AI assisted coding> & "C:/Users/MOHAMMED SHANAWAZ/AppData/Local/Programs/Python/Python313/python.exe" "d:/AI as
sisted coding/end lab exam.py"
  {'severity': 'critical', 'message': 'db down', 'source': 'db1', 'id': 'ALRT-1764066517-0', 'received_at': '2025-11-
25 15:58:37'}
  {'severity': 'high', 'message': 'cpu high', 'source': 'svc-b', 'id': 'ALRT-1764066517-2', 'received_at': '2025-11-2
5 15:58:37'}
Routes: ['db_channel:ALRT-1764066517-0', 'email_channel:ALRT-1764066517-2']
PS D:\AI assisted coding>
```

## Observation:

- The original alert pipeline was **long, mixed, and hard to read**.
- After modularizing, each step (filter, enrich, dedupe, route) is in a **separate function**, so the code becomes **clean and easy to maintain**.

- Docstrings and comments make the code **self-explanatory**, so new developers can understand it quickly.
- Testing becomes easier because each function can be tested **individually**.
- Overall, the pipeline is now **clear, organized, and more maintainable**.