```python
# main.py (skeleton, put under src/cjtrade or project root)
import asyncio
import signal
import logging
from datetime import datetime, timedelta

from cjtrade.modules.stockdata import fetch_data as PA
from cjtrade.modules.account import access as AA
from cjtrade.modules.analysis import indicator as Analysis
from cjtrade.modules.llm import suggestions as AISugg
from cjtrade.modules.candidate import manager as CandidateMgr
from cjtrade.modules.executor import executor as Executor
from cjtrade.modules.db import database as DB
from cjtrade.modules.notification import notifier as Notifier

log = logging.getLogger("cjtrade.main")
logging.basicConfig(level=logging.INFO)

# CONFIG (tune these)
PRICE_INTERVAL_SECONDS = 60        # price fetch interval (for daily/1min strategies se
DECISION_INTERVAL_SECONDS = 30     # fusion / staging interval
INVENTORY_UPDATE_SECONDS = 300     # update holdings backup
HEALTHCHECK_INTERVAL_SECONDS = 15

# queues for intra-process communication
price_queue = asyncio.Queue(maxsize=100)       # snapshots from PA -> consumed by Strat
signal_queue = asyncio.Queue(maxsize=100)      # signals from Strategy -> DecisionFusio
order_staging_queue = asyncio.Queue()          # for Executor

# graceful shutdown flag
SHUTDOWN = False

async def price_fetcher_loop():
    """Periodic fetch price snapshots and push to price_queue and DB."""
    while not SHUTDOWN:
        try:
            symbols = CandidateMgr.get_tracked_symbols()  # inventory + candidate pool
```

```
            snapshots = PA.GetPriceData(symbols)          # should be quick; if heavy,
            DB.save_snapshots(snapshots)
            # push to queue non-blocking
            try:
                price_queue.put_nowait((datetime.utcnow(), snapshots))
            except asyncio.QueueFull:
                log.warning("price_queue full, dropping snapshot")
        except Exception as e:
            log.exception("price_fetcher error: %s", e)
            Notifier.alert("price_fetcher error", str(e))
        await asyncio.sleep(PRICE_INTERVAL_SECONDS)

async def strategy_loop():
    """Consume price snapshots, generate strategy signals."""
    while not SHUTDOWN:
        ts, snapshots = await price_queue.get()
        try:
            # for each snapshot run technical rules
            signals = []
            for sym, snap in snapshots.items():
                sig = Analysis.run_technical_rules(sym, snap)  # returns None or dict(s
                if sig:
                    signals.append(sig)
            # push signals to fusion queue
            for s in signals:
                await signal_queue.put(s)
        except Exception as e:
            log.exception("strategy_loop error: %s", e)
        finally:
            price_queue.task_done()

async def decision_fusion_loop():
    """Consume signals and AI scores, perform fusion, push to staging or direct exec.""
    while not SHUTDOWN:
        try:
            sig = await signal_queue.get()
            sym = sig['symbol']
            tech_score = sig.get('score', 0.0)
```

```
            ai_score = DB.get_latest_ai_score(sym) or 0.0
            flow_score = DB.get_flow_score(sym) or 0.0

            final_score = DecisionFusion.compute(tech_score, ai_score, flow_score)
            if DecisionFusion.should_auto_execute(final_score):
                # small auto-eecution permitted
                order = DecisionFusion.form_order(sig, final_score)
                await Executor.execute_order(order, auto=True)  # executor will place a
            else:
                staging = DecisionFusion.form_staging_order(sig, final_score)
                DB.insert_order_staging(staging)
                Notifier.notify_pending_order(staging)
            signal_queue.task_done()
        except Exception as e:
            log.exception("decision_fusion error: %s", e)
            await asyncio.sleep(1)

async def inventory_update_loop():
    """Periodically refresh inventory from broker (event-driven preferred)."""
    while not SHUTDOWN:
        try:
            inv = AA.fetch_inventory()  # sync or async depending on your wrapper
            DB.save_inventory(inv)
        except Exception as e:
            log.exception("inventory_update error: %s", e)
            Notifier.alert("inventory_update error", str(e))
        await asyncio.sleep(INVENTORY_UPDATE_SECONDS)

async def healthcheck_loop():
    while not SHUTDOWN:
        # check heartbeats, DB connection, broker connection
        try:
            # healthy = DB.healthcheck() and AA.is_connected()
            healthy = DB.healthcheck() and AA.is_connected()
            if not healthy:
                Notifier.alert("Healthcheck failed")
        except Exception as e:
            log.exception("healthcheck error: %s", e)
```

```
        await asyncio.sleep(HEALTHCHECK_INTERVAL_SECONDS)

async def schedule_aicrawl():
    """Run AISuggestion tasks at scheduled times (pre/post market) in background worker
    # This can also be delegated to an external scheduler like celery beat
    while not SHUTDOWN:
        now = datetime.now()
        # Example: if time is 08:00 run premarket, 16:30 run postmarket
        if now.hour == 8 and now.minute == 0:
            asyncio.create_task(AISugg.run_pre_market())
        if now.hour == 16 and now.minute == 30:
            asyncio.create_task(AISugg.run_post_market())
        await asyncio.sleep(30)

def _signal_handler(sig):
    global SHUTDOWN
    log.info("received signal %s, shutting down...", sig)
    SHUTDOWN = True

async def main():
    # register signal handlers
    loop = asyncio.get_running_loop()
    for s in (signal.SIGINT, signal.SIGTERM):
        loop.add_signal_handler(s, lambda s=s: _signal_handler(s))

    # start background tasks
    tasks = [
        asyncio.create_task(price_fetcher_loop(), name="price_fetcher"),
        asyncio.create_task(strategy_loop(), name="strategy"),
        asyncio.create_task(decision_fusion_loop(), name="decision_fusion"),
        asyncio.create_task(inventory_update_loop(), name="inventory_update"),
        asyncio.create_task(healthcheck_loop(), name="healthcheck"),
        asyncio.create_task(schedule_aicrawl(), name="scheduler_aicrawl"),
    ]

    # wait until shutdown requested
    while not SHUTDOWN:
        await asyncio.sleep(1)
```

```
    # graceful shutdown: cancel tasks and wait
    for t in tasks:
        t.cancel()
    await asyncio.gather(*tasks, return_exceptions=True)
    # final flush and logout
    DB.close()
    AA.logout()
    log.info("shutdown complete")

if __name__ == "__main__":
    asyncio.run(main())
```