Task Execution Planner — Estrutura Refatorada

> Esqueleto de pastas/arquivos + implementações mínimas e pontos de extensão. Copie estes trechos para seus respectivos arquivos (ou use como base para colar no projeto).

📁 Estrutura Sugerida

```
streamlit_extension/
  api/
    planner_api.py
  migrations/
    001_add_columns.sql
    002_indexes.sql
  models/
    task_models.py
    scoring.py
  repos/
    tasks_repo.py
    deps_repo.py
  services/
    planner_service.py
    validation_service.py
  utils/
    db.py
    graph_algorithms.py
  tests/
    test_graph_algorithms.py
    test_scoring.py
    test_planner_service.py
```

---

migrations/001_add_columns.sql

```sql
-- Adiciona colunas explícitas (proteja em código para evitar duplicidade)
ALTER TABLE framework_tasks ADD COLUMN tdd_order INTEGER;
ALTER TABLE framework_tasks ADD COLUMN task_type VARCHAR(20) DEFAULT
'implementation';
ALTER TABLE framework_tasks ADD COLUMN priority INTEGER DEFAULT 3;
ALTER TABLE framework_tasks ADD COLUMN estimate_minutes INTEGER;
ALTER TABLE framework_tasks ADD COLUMN story_points INTEGER;
ALTER TABLE framework_tasks ADD COLUMN task_group TEXT;
ALTER TABLE framework_tasks ADD COLUMN task_sequence INTEGER;

-- Popular tdd_order com base em tdd_phase
UPDATE framework_tasks SET tdd_order = 1 WHERE tdd_phase = 'red';
UPDATE framework_tasks SET tdd_order = 2 WHERE tdd_phase = 'green';
UPDATE framework_tasks SET tdd_order = 3 WHERE tdd_phase = 'refactor';
```

migrations/002_indexes.sql

```
CREATE INDEX IF NOT EXISTS idx_tasks_epic_key ON framework_tasks(epic_id,
task_key);
CREATE INDEX IF NOT EXISTS idx_td_task ON task_dependencies(task_id);
CREATE INDEX IF NOT EXISTS idx_td_depkey ON
task_dependencies(depends_on_task_key);
```

---

models/task_models.py

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Optional, Any, Dict

@dataclass
class Task:
    id: int
    task_key: str
    epic_id: int
    title: Optional[str] = None
    description: Optional[str] = None
    tdd_phase: Optional[str] = None  # 'red'|'green'|'refactor'|None
    tdd_order: Optional[int] = None  # 1,2,3
    task_type: str = 'implementation'  # 'analysis'|'implementation'|...
    status: Optional[str] = None
    estimate_minutes: Optional[int] = None
    story_points: Optional[int] = None
    priority: int = 3                 # 1 crítico .. 5 backlog
    task_group: Optional[str] = None
    task_sequence: Optional[int] = None
    created_at: Optional[str] = None

    @classmethod
    def from_db_row(cls, row: Dict[str, Any]) -> "Task":
        return cls(
            id=row["id"], task_key=row["task_key"], epic_id=row["epic_id"],
            title=row.get("title"), description=row.get("description"),
            tdd_phase=row.get("tdd_phase"), tdd_order=row.get("tdd_order"),
            task_type=row.get("task_type", "implementation"),
            status=row.get("status"),
            estimate_minutes=row.get("estimate_minutes"),
            story_points=row.get("story_points"),
            priority=row.get("priority", 3),
            task_group=row.get("task_group"),
```

```python
task_sequence=row.get("task_sequence"),
            created_at=row.get("created_at"),
        )

    @property
    def is_tdd_task(self) -> bool:
        return self.tdd_order in (1, 2, 3)

    @property
    def effort(self) -> int:
        # fallback coerente
        return max(self.estimate_minutes or self.story_points or 1, 1)

@dataclass
class TaskDependency:
    id: int
    task_id: int
    depends_on_task_key: str
    depends_on_task_id: Optional[int] = None
    dependency_type: Optional[str] = None

    @classmethod
    def from_db_row(cls, row: Dict[str, Any]) -> "TaskDependency":
        return cls(
            id=row["id"], task_id=row["task_id"],
            depends_on_task_key=row["depends_on_task_key"],
            depends_on_task_id=row.get("depends_on_task_id"),
            dependency_type=row.get("dependency_type"),
        )

@dataclass
class TaskPriorityScore:
    task_key: str
    total_score: float
    priority_score: float
    value_density_score: float
    unblock_score: float
    critical_path_score: float
    tdd_bonus_score: float
    aging_score: float

@dataclass
class TaskExecutionResult:
    epic_id: int
    execution_order: list[str]
    parallel_batches: list[list[str]]
    total_tasks: int
    tasks_with_dependencies: int
    total_dependencies: int
    estimated_total_minutes: int
    critical_path_length: int
    warnings: list[str]
    errors: list[str]

class TaskModelError(Exception):
    ...

class CyclicDependencyError(Exception):
    ...

class TaskNotFoundError(Exception):
    ...
```

models/scoring.py

```
from __future__ import annotations
from typing import Dict, Set
from .task_models import Task, TaskPriorityScore

# Pesos padrão (tunable)
W_PRIO = 10.0
W_DENS = 6.0
W_UNBLOCK = 3.0
W_CRIT = 2.0
W_TDD = 1.0
W_AGING = 0.2

def tdd_bonus(task: Task) -> float:
    if task.tdd_order in (1, 2, 3):
        return {1: 3.0, 2: 2.0, 3: 1.0}[task.tdd_order]
    return 0.0

def value_density(task: Task) -> float:
    # Exemplo simples: prioridade invertida / esforço
    prio = max(1, min(5, task.priority or 3))
    return (6 - prio) / task.effort

def calc_scores(
    tasks: list[Task],
    adjacency: Dict[str, Set[str]],
    critical_time: Dict[str, int],
    critical_nodes: set[str],
) -> Dict[str, TaskPriorityScore]:
    out = {}
    max_crit = max(critical_time.values()) if critical_time else 1
    for t in tasks:
        dens = value_density(t)
        prio_score = (6 - (t.priority or 3))
        unblock = len(adjacency.get(t.task_key, set()))
        crit_score = (critical_time.get(t.task_key, 0) / max_crit) * 10.0
        tdd = tdd_bonus(t)
        aging = 0.0  # TODO: calcular por dias
        total = (
            W_PRIO*prio_score + W_DENS*dens + W_UNBLOCK*unblock +
            W_CRIT*crit_score + W_TDD*tdd + W_AGING*aging
        )
        out[t.task_key] = TaskPriorityScore(
            task_key=t.task_key,
            total_score=total,
            priority_score=prio_score,
            value_density_score=dens,
            unblock_score=float(unblock),
            critical_path_score=crit_score,
            tdd_bonus_score=tdd,
            aging_score=aging,
        )
    return out

def priority_tuple(task: Task, score: TaskPriorityScore) -> tuple:
    # heapq min-heap → inverter sinal do score; tie-breaks estáveis
    return (-score.total_score, -(6 - (task.priority or 3)), task.effort,
task.task_key)
```

---

repos/tasks_repo.py

```
from __future__ import annotations
from typing import List
from ..utils.db import dict_rows
from ..models.task_models import Task

class TasksRepo:
    def __init__(self, db):
        self.db = db

    def list_by_epic(self, epic_id: int) -> List[Task]:
        sql = """
            SELECT id, task_key, epic_id, title, description,
                   tdd_phase, tdd_order, task_type, status,
                   estimate_minutes, story_points, priority,
                   task_group, task_sequence, created_at
            FROM framework_tasks
            WHERE epic_id = ? AND deleted_at IS NULL
            ORDER BY task_key
        """
        with dict_rows(self.db):
            rows = self.db.execute(sql, (epic_id,)).fetchall()
        return [Task.from_db_row(dict(r)) for r in rows]
```

repos/deps_repo.py

```
from __future__ import annotations
from typing import List
from ..utils.db import dict_rows
from ..models.task_models import TaskDependency

class DepsRepo:
    def __init__(self, db):
        self.db = db

    def list_by_epic(self, epic_id: int) -> List[TaskDependency]:
        sql = """
            SELECT td.id, td.task_id, td.depends_on_task_key,
                   td.depends_on_task_id, td.dependency_type
            FROM task_dependencies td
            JOIN framework_tasks ft ON td.task_id = ft.id
            WHERE ft.epic_id = ?
            ORDER BY td.id
        """
        with dict_rows(self.db):
            rows = self.db.execute(sql, (epic_id,)).fetchall()
        return [TaskDependency.from_db_row(dict(r)) for r in rows]
```

---

utils/db.py

```
from __future__ import annotations
from contextlib import contextmanager
import sqlite3

@contextmanager
def dict_rows(conn: sqlite3.Connection):
    old = conn.row_factory
    conn.row_factory = sqlite3.Row
    try:
        yield
    finally:
        conn.row_factory = old
```

utils/graph_algorithms.py

```
from __future__ import annotations
from typing import Dict, Set, List, Callable
import heapq
```

```
# ---- Ciclos / DAG ----

def validate_dag(adj: Dict[str, Set[str]]):
    WHITE, GRAY, BLACK = 0,1,2
    color = {u:WHITE for u in adj}
    parent = {}

    def dfs(u):
        color[u] = GRAY
        for v in adj[u]:
            if color[v] == GRAY:
                # reconstrói caminho
                path = [v, u]
                p = parent.get(u)
                while p and p != v:
                    path.append(p); p = parent.get(p)
                path.append(v)
                return False, list(reversed(path))
            if color[v] == WHITE:
                parent[v] = u
                ok, cyc = dfs(v)
                if not ok: return ok, cyc
        color[u] = BLACK
        return True, None

    for u in list(adj):
        if color[u] == WHITE:
            ok, cyc = dfs(u)
            if not ok: return False, cyc
    return True, None

# ---- Topo com prioridade ----

def topo_with_priority(adj: Dict[str, Set[str]], score_tuple: Callable[[str],
tuple]) -> List[str]:
    indeg = {u:0 for u in adj}
    for u in adj:
        for v in adj[u]:
            indeg[v] = indeg.get(v, 0) + 1
    heap = []  # min-heap com tuplas já invertidas
    for u,d in indeg.items():
        if d == 0:
            heapq.heappush(heap, score_tuple(u))
    order = []
    while heap:
        *_, u = heapq.heappop(heap)
        order.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                heapq.heappush(heap, score_tuple(v))
    if len(order) != len(indeg):
        missing = set(indeg) - set(order)
        raise RuntimeError(f"Ordenação incompleta; verifique deps:
{sorted(missing)}")
    return order

# ---- Longest path ponderado (DAG) ----

def longest_path_weighted(adj: Dict[str, Set[str]], weight: Dict[str, int]):
    # topo simples
    indeg = {u:0 for u in adj}
    for u in adj:
        for v in adj[u]:
            indeg[v] = indeg.get(v, 0) + 1
    q = [u for u,d in indeg.items() if d == 0]
    from collections import deque
    dq = deque(q)
    topo = []
    while dq:
        u = dq.popleft()
        topo.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                dq.append(v)
    if len(topo) != len(adj):
        return {}
    dist = {u: weight.get(u,1) for u in adj}
    for u in topo:
        for v in adj[u]:
            cand = dist[u] + weight.get(v,1)
            if cand > dist[v]:
                dist[v] = cand
    return dist

def find_critical_path_nodes(adj: Dict[str, Set[str]], weight:
Dict[str,int]):

# Heurística: nós com maior dist (top-k) → aqui devolvemos todos no maior
dist
    dist = longest_path_weighted(adj, weight)
    if not dist:
        return []
    m = max(dist.values())
    return [u for u,d in dist.items() if d == m]

# ---- Batches por níveis com prioridade externa ----

def batches_by_levels(adj: Dict[str, Set[str]], priority_rank: Dict[str,int],
max_parallel: int = 999) -> List[List[str]]:
    indeg = {u:0 for u in adj}
    for u in adj:
        for v in adj[u]:
            indeg[v] = indeg.get(v, 0) + 1
    ready = [u for u,d in indeg.items() if d == 0]
    ready.sort(key=lambda k: priority_rank.get(k, 10**9))
    batches = []
    while ready:
        batch = ready[:max_parallel]
        batches.append(batch)
        next_ready = ready[max_parallel:]
        for u in batch:
            for v in adj[u]:
                indeg[v] -= 1
                if indeg[v] == 0:
                    next_ready.append(v)
        # ordena por ranking (menor é melhor)
        # dedup preservando ordem
        seen=set(); tmp=[]
        for x in next_ready:
            if x not in seen:
                seen.add(x); tmp.append(x)
        tmp.sort(key=lambda k: priority_rank.get(k, 10**9))
        ready = tmp
    return batches
```

---

services/planner_service.py

```
from __future__ import annotations
from typing import Dict, Set
from collections import defaultdict
from ..models.task_models import Task, TaskDependency, TaskExecutionResult
from ..models.scoring import calc_scores, priority_tuple
from ..utils.graph_algorithms import validate_dag, topo_with_priority,
longest_path_weighted, find_critical_path_nodes, batches_by_levels
from ..repos.tasks_repo import TasksRepo
from ..repos.deps_repo import DepsRepo

class TaskExecutionPlanner:
    def __init__(self, db_conn):
        self.tasks_repo = TasksRepo(db_conn)
        self.deps_repo = DepsRepo(db_conn)

    def _build_adjacency(self, tasks: list[Task], deps: list[TaskDependency])
-> Dict[str, Set[str]]:
        keys = {t.task_key for t in tasks}
        adj = {t.task_key: set() for t in tasks}
        # deps explícitas
        for d in deps:
            # A -> B (A antes de B)
            dep = d.depends_on_task_key
            # pegar B via task_id
            # aqui assumimos que o repos trouxe somente deps válidas do épico
            # se necessário, mapear id->key
            # Para simplicidade, adicione apenas se ambas keys existirem no
épico
            # (se precisar de id→key, traga no SELECT do repo)
            # Usaremos um fallback simples:

# d.task_id não está com key — então deixe explícito via repos se necessário
            pass
        # OBS: como d.task_id não tem key aqui, recomendamos ajustar o
DepsRepo
        # para retornar também a task_key dependente. Implemento abaixo a
versão correta.
        return adj

    def _build_adjacency_correct(self, tasks: list[Task], deps:
list[TaskDependency]) -> Dict[str, Set[str]]:
        id_to_key = {t.id: t.task_key for t in tasks}
        keys = set(id_to_key.values())
        adj = {t.task_key: set() for t in tasks}
        for d in deps:
            src = d.depends_on_task_key
            tgt = id_to_key.get(d.task_id)
            if src in keys and tgt in keys:
                adj[src].add(tgt)
        # Arestas implícitas TDD: RED->GREEN->REFACTOR por group
        groups = defaultdict(list)
        for t in tasks:
            if t.task_group and t.tdd_order:
                groups[t.task_group].append(t)
        for g, arr in groups.items():
            arr.sort(key=lambda x: x.tdd_order)
            for a, b in zip(arr, arr[1:]):
                adj[a.task_key].add(b.task_key)
        return adj

    def plan_epic_execution(self, epic_id: int) -> TaskExecutionResult:
        tasks = self.tasks_repo.list_by_epic(epic_id)
        deps = self.deps_repo.list_by_epic(epic_id)
        if not tasks:
            return TaskExecutionResult(
                epic_id=epic_id, execution_order=[], parallel_batches=[],
                total_tasks=0, tasks_with_dependencies=0,
total_dependencies=0,
                estimated_total_minutes=0, critical_path_length=0,
                warnings=["Épico não possui tarefas"], errors=[]
            )
        adj = self._build_adjacency_correct(tasks, deps)
        ok, cyc = validate_dag(adj)
        if not ok:
            raise RuntimeError(f"Ciclo detectado: {' -> '.join(cyc)}")
        weight = {t.task_key: t.effort for t in tasks}
        crit_time = longest_path_weighted(adj, weight)
        crit_nodes = set(find_critical_path_nodes(adj, weight))
        scores = calc_scores(tasks, adj, crit_time, crit_nodes)
        # Score tuple provider
        def s_tuple(u: str):
            t_map = {t.task_key: t for t in tasks}
            return priority_tuple(t_map[u], scores[u])
        order = topo_with_priority(adj, s_tuple)
        rank = {k:i for i,k in enumerate(order)}
        batches = batches_by_levels(adj, rank, max_parallel=5)
        total_deps = sum(len(v) for v in adj.values())
        with_deps = len({v for vs in adj.values() for v in vs})
        est_total = sum(t.effort for t in tasks)
        max_crit = max(crit_time.values()) if crit_time else 0
        warns = []
        if total_deps == 0:
            warns.append("Sem dependências; tudo pode rodar em paralelo")
        return TaskExecutionResult(
            epic_id=epic_id,
            execution_order=order,
            parallel_batches=batches,
            total_tasks=len(tasks),
            tasks_with_dependencies=with_deps,
            total_dependencies=total_deps,
            estimated_total_minutes=est_total,
            critical_path_length=max_crit,
            warnings=warns,
            errors=[]
        )
```

services/validation_service.py

```
from __future__ import annotations
from typing import Dict, Set, Any
from ..utils.graph_algorithms import validate_dag

class ValidationService:
    def validate(self, adjacency: Dict[str, Set[str]]) -> dict[str, Any]:
        ok, cyc = validate_dag(adjacency)
        return {"is_dag": ok, "cycle": cyc}
```

---

api/planner_api.py

```python
from __future__ import annotations
from ..services.planner_service import TaskExecutionPlanner

# Cola leve para Streamlit ou outra UI

def plan_epic(db_conn, epic_id: int):
    planner = TaskExecutionPlanner(db_conn)
    return planner.plan_epic_execution(epic_id)
```

---

tests/test_graph_algorithms.py

```python
from streamlit_extension.utils.graph_algorithms import validate_dag,
topo_with_priority, longest_path_weighted

def test_simple_topo():
    adj = {"A": {"B"}, "B": {"C"}, "C": set()}
    rank = {"A":0, "B":1, "C":2}
    def tup(u): return (0, 0, 0, u)
    order = topo_with_priority(adj, tup)
    assert order == ["A","B","C"]

def test_longest_path_weighted():
    adj = {"A": {"B"}, "B": {"C"}, "C": set()}
    w = {"A":1, "B":2, "C":3}
    dist = longest_path_weighted(adj, w)
    assert dist["C"] == 1+2+3
```

tests/test_scoring.py

```python
from streamlit_extension.models.task_models import Task
from streamlit_extension.models.scoring import value_density, tdd_bonus

def test_value_density_monotonic():
    t1 = Task(id=1, task_key="A", epic_id=1, priority=1, estimate_minutes=10)
    t2 = Task(id=2, task_key="B", epic_id=1, priority=5, estimate_minutes=10)
    assert value_density(t1) > value_density(t2)

def test_tdd_bonus():
    assert tdd_bonus(Task(id=1, task_key="x", epic_id=1, tdd_order=1)) >
tdd_bonus(Task(id=2, task_key="y", epic_id=1, tdd_order=3))
```

tests/test_planner_service.py

```python
from streamlit_extension.services.planner_service import TaskExecutionPlanner
from streamlit_extension.models.task_models import Task, TaskDependency

class FakeRepoPlanner(TaskExecutionPlanner):
    def __init__(self, tasks, deps):
        self.tasks = tasks
        self.deps = deps
    def plan(self):
        adj = self._build_adjacency_correct(self.tasks, self.deps)
        ok, cyc = True, None
        assert ok
```

Notas finais

- **Repos**: se quiser `deps_repo` já trazendo `dependent_task_key`, ajuste o SELECT (JOIN adicional) e simplifique `_build_adjacency_correct`.
- **Pesos/heurística**: estão isolados em `models/scoring.py`. Tune à vontade.
- **Arestas TDD**: dependem de `task_group` + `tdd_order` — já contemplado.
- **Erros**: topo sort falha duro em inconsistência; trate na UI conforme necessidade.