Core Idea: The model tries to find the *best* start time (`s`) for *each* task (`i`) by setting exactly one `X[i, s]` variable to 1 for each task `i`. Constraints are the rules that define what makes a schedule *valid* or *feasible*.

---

1. Task Assignment Constraint

- **Intuition:** Every single task you provide *must* be scheduled exactly once. You can't skip a task, nor can you schedule the same task multiple times.
- **Mathematical Model:** `∑ (from s=0 to S_total-1) X[i,s] = 1` for each task `i`.
  - This sums up the `X[i,s]` variable (which is 0 or 1) over *all possible start slots* `s` for a *single task* `i`.
  - By setting the sum equal to 1, it forces the model to choose exactly *one* slot `s` where `X[i,s]` will be 1 (meaning task `i` starts there), and all other `X[i,s]` for that task `i` must be 0.
- **Code Implementation:**
  - **PuLP:** `model += lpSum(X[(i, s)] for s in range(TOTAL_SLOTS)) == 1, f"Assign_{task_key}"`
  - **Gurobi:** `m.addConstrs((X.sum(i, '*') == 1 for i in range(n_tasks)), name="Assign")`
- **Example:** If you have Task 'A', the model *must* find some slot `s` (say, slot 50) such that `X['A', 50] = 1`, and for all other slots `s'` (0-49, 51-391), `X['A', s'] = 0`.

---

2. Deadlines and Horizon Constraint

- **Intuition:** Tasks must finish *before* their deadline. Also, tasks must fit *entirely* within the 7-day schedule window (they can't start so late that they spill over past the last slot).
- **Mathematical Model:** `X[i,s] = 0` if starting task `i` at slot `s` would violate either the deadline or the horizon boundary.
  - **Deadline Check:** If task `i` starts at `s` and has duration `dur_i`, its last occupied slot is `s + dur_i - 1`. If this last slot index is *greater than* the deadline slot `dl_i`, then starting at `s` is forbidden (`X[i,s] = 0`).
  - **Horizon Check:** If task `i` starts at `s`, it needs `dur_i` slots. The index of the slot *after* it finishes is `s + dur_i`. If this index is *greater than* the total number of slots `S_total` (meaning it goes beyond the last valid slot index `S_total - 1`), then starting at `s` is forbidden (`X[i,s] = 0`).
- **Code Implementation:**
  - **PuLP:** Loops through `s` and adds `model += (X[(i, s)] == 0)` if the `if s + dur - 1 > dl:` or `if s + dur > TOTAL_SLOTS:` condition is met.
  - **Gurobi:** Loops through `s` and adds `m.addConstr(X[i, s] == 0)` if the same conditions are met.
- **Example:**
  - Task 'B' lasts 4 slots (`dur_B = 4`) and deadline `dl_B = 100`. Trying to start at `s = 98`: Task occupies slots 98, 99, 100, 101. Last slot is 101. Since `101 > 100`, this start is invalid. Constraint forces `X['B', 98] = 0`. Starting at `s = 97` (occupies 97, 98, 99, 100) is okay regarding the deadline.
  - `TOTAL_SLOTS = 392`. Task 'C' lasts 2 slots (`dur_C = 2`). Trying to start at `s = 391`: Task would need slots 391 and 392. Slot 392 doesn't exist. `s + dur_C = 391 + 2 = 393`. Since `393 > 392`, this start is invalid. Constraint forces `X['C', 391] = 0`.

---

3. No Overlap Constraint

- **Intuition:** You can only do one scheduled task at a time in any given 15-minute slot. No double-booking!
- **Mathematical Model:** `∑ (over all tasks i) ∑ (from s = max(0, t - dur_i + 1) to t) X[i,s] ≤ 1` for each time slot `t`.
  - For a specific time slot `t`, we need to check which tasks *could potentially be running*.
  - A task `i` (duration `dur_i`) is running *during* slot `t` if it started at some slot `s` such that `s <= t` (it started at or before `t`) AND `t <= s + dur_i - 1` (it hasn't finished before `t`). This range of possible start times `s` is exactly `[max(0, t - dur_i + 1), t]`.
  - The constraint sums up the `X[i,s]` variables for *all* task/start-time combinations that would result in occupation of slot `t`. Since each `X[i,s]` is 0 or 1, this sum counts how many tasks are scheduled to be active in slot `t`.
  - By forcing this sum to be `<= 1`, we ensure that at most one task can be active in slot `t`.
- **Code Implementation:**
  - Builds a list `occupying_tasks_vars` containing `X[(i, s)]` for all `i` and relevant `s` that cover slot `t`.
  - **PuLP:** `model += lpSum(occupying_tasks_vars) <= 1, f"NoOverlap_s{t}"`
  - **Gurobi:** `m.addConstr(gp.quicksum(occupying_tasks_vars) <= 1, name=f"NoOverlap_s{t}")`
- **Example:** Consider slot `t = 20`.
  - Task 'D' (dur=3) covers slot 20 if it starts at s=18, 19, or 20.
  - Task 'E' (dur=2) covers slot 20 if it starts at s=19 or 20.
  - The constraint for `t=20` ensures: `X['D', 18] + X['D', 19] + X['D', 20] + X['E', 19] + X['E', 20] <= 1`.
  - This prevents solutions where, for example, `X['D', 19] = 1` (Task D running in 19, 20, 21) AND `X['E', 20] = 1` (Task E running in 20, 21) because both would need slot 20.

---

4. Preferences Constraint

- **Intuition:** If a user says they prefer to do a task in the "morning", the schedule should only place its *start time* in a morning slot.
- **Mathematical Model:** `X[i,s] = 0` for any slot `s` that is *not* in the `AllowedSlots_i` set defined by the task's preference.
  - `AllowedSlots_i` is pre-calculated (e.g., all slots from 8am-12pm for "morning" across all 7 days).
  - This constraint simply forbids starting task `i` at any slot `s` outside its designated preference window.
- **Code Implementation:**
  - Looks up the set of allowed slots based on `tasks[i]['preference']`.
  - Loops through all `s` and if `s` is *not in* `allowed_slots`:
    - **PuLP:** `model += X[(i, s)] == 0, f"PrefWin_{task_key}_s{s}"`
    - **Gurobi:** `m.addConstr(X[i, s] == 0, name=f"PrefWin_{task_key}_s{s}")`
- **Example:** Task 'F' has preference "afternoon" (e.g., slots 12pm-4pm, like index 16-31 for Day 0). Slot `s = 10` is a morning slot. This constraint forces `X['F', 10] = 0`. Task 'F' cannot start at slot 10.

---

5. Commitments Constraint

- **Intuition:** Tasks cannot overlap with fixed commitments (like meetings or appointments) that block certain slots.
- **Mathematical Model:** `X[i,s] = 0` if the time interval occupied by task `i` starting at `s` (slots `{s, s+1, ..., s + dur_i - 1}`) intersects with the set of committed slots `C`.
    - It checks if *any* of the slots the task would use are already marked as committed.
    - If there's any overlap (`intersection != empty set`), starting task `i` at `s` is forbidden.
- **Code Implementation:**
    - Calculates the `task_occupies` set of slots for task `i` starting at `s`.
    - Checks if `task_occupies.intersection(committed_slots)` is non-empty.
    - If it is, it adds the constraint:
        - **PuLP:** `model += X[(i, s)] == 0, f"CommitOverlap_{task_key}_s{s}"`
        - **Gurobi:** `m.addConstr(X[i, s] == 0, name=f"CommitOverlap_{task_key}_s{s}")`
- **Example:** Slot `s = 60` is a commitment (`60 in C`). Task 'G' has `dur_G = 3`.
    - Try starting at `s = 58`. Occupies {58, 59, 60}. Intersects with {60}. Invalid start. Force `X['G', 58] = 0`.
    - Try starting at `s = 59`. Occupies {59, 60, 61}. Intersects with {60}. Invalid start. Force `X['G', 59] = 0`.
    - Try starting at `s = 60`. Occupies {60, 61, 62}. Intersects with {60}. Invalid start. Force `X['G', 60] = 0`.
    - Try starting at `s = 61`. Occupies {61, 62, 63}. No intersection with {60}. This start is allowed (by this specific constraint).

---

6. Leisure Calculation and Occupation Link (Y)

- **Intuition:** We want to calculate leisure time for the objective function. Leisure only happens in 15-minute slots that are *completely free* – meaning they are *not* committed AND *not* occupied by any scheduled task. We use an intermediate variable `Y[s]` to track task occupation.
- **Mathematical Model & Code:** This is done in a few linked parts:
    - **(Link Y to X):** `Y[s] = ∑ (over tasks i) ∑ (over starts s' covering s) X[i,s']`
        - **Intuition:** `Y[s]` should be 1 if *any* task occupies slot `s`, and 0 otherwise.
        - **Model:** This uses the *exact same sum* as the "No Overlap" constraint. Since that sum is forced to be 0 or 1, setting `Y[s]` equal to it correctly captures the occupation status.
        - **Code (PuLP):** `model += Y[s] == lpSum(occupying_task_vars), f"Link_Y_Exact_{s}"`
        - **Code (Gurobi):** `m.addConstr(Y[s] == occupying_task_vars_sum, name=f"Link_Y_Exact_{s}")`
    - **(Calculate L based on Y and Commitments):**
        - `L[s] = 0` if slot `s` is committed (`s in C`). (Easy: Code sets `L_var[s] == 0`)
        - `L[s] <= 15 * (1 - Y[s])` if slot `s` is *not* committed (`s not in C`).
            - **Intuition:** If not committed: If a task occupies the slot (`Y[s]=1`), then `1 - Y[s] = 0`, so `L[s] <= 0`. Since leisure can't be negative, `L[s]` becomes 0. If no task occupies the slot (`Y[s]=0`), then `1 - Y[s] = 1`, so `L[s] <= 15`. The objective function (maximizing leisure) will push `L[s]` to 15.
            - **Code (PuLP):** `model += L_var[s] <= 15 * (1 - Y[s]), f"LeisureBound_NotCommitted_{s}"`
            - **Code (Gurobi):** `m.addConstr(L_var[s] <= 15 * (1 - Y[s]), name=f"LeisureBound_NotCommitted_{s}")`
- **Example:** Consider slot `s = 70`.
    - Case 1: `70 in C` (committed). `L[70]` is forced to 0.
    - Case 2: `70 not in C`. Task 'H' starts at `s=70`, `dur_H=1`. So `X['H', 70]=1`. `Y[70]` becomes 1. `L[70] <= 15 * (1-1) = 0`. So `L[70]=0`.
    - Case 3: `70 not in C`. No task occupies slot 70. `Y[70]` becomes 0. `L[70] <= 15 * (1-0) = 15`. Objective makes `L[70]=15`.

---

7. Daily Limits Constraint (Optional)

- **Intuition:** Prevent scheduling too much work on any single day. Limit the *total number of 15-minute slots* used by tasks within each 24-hour period (from 8am to 10pm).
- **Mathematical Model:** `∑ (for s in Day d) Y[s] ≤ Limit_daily` for each day `d`.
    - It sums the `Y[s]` variables (which are 1 if a task occupies the slot, 0 otherwise) for all slots `s` belonging to a specific day `d`.
    - This sum represents the total count of task-occupied slots on that day.
    - It forces this total count to be no more than the `Limit_daily`.
- **Code Implementation:**
    - Calculates the sum of `Y[s]` for the day's slots (`day_start_slot` to `day_end_slot - 1`).
    - **PuLP:** `model += daily_task_slots_sum <= daily_limit_slots, f"DailyLimit_Day_{d}"`
    - **Gurobi:** `m.addConstr(daily_task_slots_sum <= daily_limit_slots, name=f"DailyLimit_Day_{d}")`
- **Example:** `Limit_daily = 20` (5 hours). On Day 1 (slots 56-111):
    - Schedule: Task 'I' (dur=10, start=60), Task 'J' (dur=12, start=80).
    - Occupied slots on Day 1: 10 slots from Task I + 12 slots from Task J = 22 slots.
    - Sum `Y[s]` for `s` =56 to 111 is 22.
    - `22 <= 20` is FALSE. This schedule violates the daily limit for Day 1 and is therefore invalid. The solver would need to find a different arrangement.

In essence, these constraints work together to carve out the space of *possible* schedules, and the objective function then guides the solver to find the *best* one within that space according to the weights given to leisure and stress.