DBCO

DBCO FREE BUILD BRIEF · TEAM OPS

The Scheduling & Shift System

A free, paste-ready build brief from the DBCO Value Library. Drop the companion **.md** file straight into Claude Code and build the system from the video yourself.

BUILD IT YOURSELF | CLAUDE CODE READY | NO SAAS FEES

A build brief for agency owners who want to stop running their team out of a spreadsheet.

Before you read: what this is

This is the free companion file to the video. It's the actual spec for the system I built to run scheduling, discipline, and shift tracking for a chatting team — the schedule builder, the penalty ledger, the clock-in/clock-out payroll engine, the watchdog crons that catch no-shows, and the Telegram notification layer that pings everyone automatically.

I cloud-coded the whole thing myself with Claude Code. No SaaS, no monthly seat fees, no "sorry that feature is on the enterprise plan." It runs on one Node/Express server, stores everything in flat JSON files, and I own every line. You can build the same thing.

This is a scaffold, not a turnkey download. I'm giving you the architecture, the data model, the algorithm, the routes, and paste-ready prompts. You drop these into Claude Code and it scaffolds the system for you. You'll still wire, test, and adapt it to your own operation. That's the point — you end up understanding it, so when it breaks at 2am you can fix it.

One thing to set expectations up front: this brief covers the **admin side and the system side** in full — the builder, the data, the algorithm, the notifications, all of it. The **chatter-facing panel** — the login your team uses to actually see their schedule and penalties — is the front-end you build on top for your own team, wired to your own auth and styled how you want your chatters to experience it. It comes up naturally as you go.

Part 1 — The problem, and why it matters

Here's the situation every agency owner hits around 8–10 chatters.

You're running a 24/7 operation. Multiple creators, multiple time zones, three shifts a day, seven days a week. You need the right person on the right creator at the right hour, every hour, or you're leaving money on the table — a dark shift on a busy creator is pure lost revenue.

At first you run it in a spreadsheet. Then it falls apart, and it falls apart in a very specific sequence:

- **Coverage gaps you don't find out about until it's too late.** Someone doesn't show up for the 2am shift. You find out at 9am when the creator's inbox has 40 unanswered messages and the fan who was about to drop $500 got bored and left.
- **Availability chaos.** You're chasing people in DMs — "what can you work next week?" — and half of them reply Sunday night when you already built the schedule Friday.
- **Scheduling by vibes.** You build the grid tired at midnight, double-book the same person on two creators, leave a night slot empty, and forget

who actually asked for what.

- **Discipline with no paper trail.** Someone's slow three weeks running. You "talked to them about it" but there's no record, no escalation, no consistency. The next person who does the same thing gets a different consequence and now you look unfair.
- **Payroll arguments.** "I worked 8 hours." "The system says 6." You have no source of truth, so you eat the difference or you fight about it.

Every one of these is a **money leak** or a **trust leak**. Coverage gaps leak money directly. Inconsistent discipline leaks trust and your best people quietly start looking elsewhere. Payroll fights leak both.

The fix is a system where:

1. Chatters submit availability by a hard deadline.
2. The schedule gets **built automatically from everyone's submitted availability** — fairly and consistently — not by whoever you remembered to assign.
3. Every shift is clocked in and clocked out, and pay is computed from real timestamps, not memory.
4. Discipline is a **ledger** — every offense recorded, auto-numbered, escalating, tied to the week and to payroll.
5. Robots watch the shifts and scream — to you and to the chatter — the moment something is off. No-show at 20 minutes. Stuck shift at 2 hours. Auto-closed at 3.

That's what we're building.

Part 2 — The architecture, plain and simple

Don't overthink the stack. Mine is deliberately boring so it never surprises me.

- **One server.** `server.js` — Node + Express, listening on a single port (mine is `3002`). A reverse proxy (I use Caddy) maps a public domain to it and handles TLS.
- **No SQL database.** All state lives in **flat JSON files** loaded and saved by tiny cached accessor functions. For a team of ~100 people and ~1000 shift records this is more than fine, it's *faster* to reason about, and your entire database is `git`-diffable and human-readable. When you outgrow it you swap the accessors for a DB and nothing else changes.
- **Static admin dashboard.** A single `admin.html` served by the same box. Vanilla HTML/JS calling the JSON API. No build step, no framework, no npm hell.
- **A separate chatter dashboard.** A second page your team logs into to see their own schedule and penalties — you build this to fit your own workflow and branding.
- **Telegram bots for notifications.** Push everything important to Telegram so nobody has to keep a tab open.

The mental model: **the server is the brain and the source of truth. The JSON files are its memory. The admin page is your window into it. Telegram is how it reaches out and taps people on the shoulder.**

The cached-accessor pattern (build this first)

Every data file gets a matched loader/saver pair with an in-memory cache so you're not hitting disk on every request. Mine look like `lC()/sC()` for the main chatters file, `lPEN()/sPEN()` for penalties, `lAV()/sAV()` for availability, and so on. The pattern is trivial:

```
let _cCache = null;
function lC() {                        // load-Chatters
  if (_cCache) return _cCache;
  try { _cCache = JSON.parse(fs.readFileSync(CHATTERS_FILE, 'utf8')); }
  catch { _cCache = { chatters: [], shifts: [], schedule: [], notificatio
  return _cCache;
}
function sC(data) {                     // save-Chatters
  _cCache = data;
  fs.writeFileSync(CHATTERS_FILE, JSON.stringify(data, null, 2));
}
```

Build one of these per file and you never think about persistence again.

Part 3 — The data model (this is 80% of the system)

Get the shapes right and the routes write themselves. Here's every file and every important field. Copy these shapes.

`chatters-data.json` — the big one

Top-level keys: `chatters[]`, `shifts[]` (worked shifts), `schedule[]` (planned assignments), `holidays`, `swap_requests[]`, `notifications[]`, `archives`, `slot_config`, `scheduler_tier_prefs`, `images`.

A `chatter`:

```
{
  "id": "uuid",
  "name": "Jane Doe",
  "username": "jane",
  "password_hash": "bcrypt/argon2 hash — never store plaintext (see secur
  "hourly_pay": 4,
  "commission_rate": 0.03,
  "discord_username": "jane#0001",
  "timezone": "Europe/Berlin",
  "active": true,
  "eth_address": "0x… (ERC-20, for payouts)",
  "tg_chat_id": "linked Telegram chat id — gates every DM to this person"
  "created_at": "ISO"
}
```

A `shift` (a *worked* record — created on clock-in, finalized on clock-out):

```
{
  "id": 1234,
  "chatter_id": "uuid", "chatter_name": "Jane Doe",
  "date": "2026-07-01",
  "shift_slot": "evening", "shift_label": "Evening 18:00–02:00",
  "shift_start": "18:03 CET", "shift_end": "02:01 CET", "hours": 7.97,
  "tips_gross": 800, "messages_gross": 1200,
  "tips_net": 640, "messages_net": 960, "total_net": 1600,
  "commission": 48, "base_pay": 31.88, "payout": 79.88,
  "bonus": 0, "penalty": 0,
  "paid": false, "paid_at": null,
  "handoff_note": "Whale in DMs, mid-negotiation on a $500 custom.",
  "status": "completed",
  "checklist": [ /* per-shift task list, see below */ ],
  "clocked_in_at": "ISO",
  "agency_id": "…", "agency_name": "…", "creator_id": "…", "creator_name"
  "force_completed": false,
  "created_at": "ISO"
}
```

A `schedule` entry (a *planned* assignment — the builder's output):

```
{
  "id": "uuid",
  "week": "2026-W27",          // ISO week
  "date": "2026-07-01",        // note the night-slot date shift, below
  "slot_id": "evening",
  "chatter_id": "uuid", "chatter_name": "Jane Doe",
  "creator_id": "…", "creator_name": "…",
  "agency_id": "…", "agency_name": "…",
  "status": "assigned",         // assigned | claimed | open
  "published": false,           // false = draft, invisible to chatters
  "built_by_scheduler": "auto", // auto | cover | manual
  "callout_reason": null, "callout_at": null,
  "original_chatter_id": null, "original_chatter_name": null,
  "claimed_at": null, "swapped_from": null, "swapped_at": null,
  "created_at": "ISO"
}
```

Note the two-table split: `schedule[]` **is the plan,** `shifts[]` **is reality.** They are deliberately decoupled. Clock-in does not require a matching schedule row (see the shift-tracking section for why).

`slot_config` — three shifts covering 24 hours. Defaults:

slot	label	start	end
`morning`	10:00–18:00	10	18
`evening`	18:00–02:00	18	2 (wraps midnight)
`night`	02:00–10:00	2	10

Make these editable from the admin — don't hardcode them.

`penalties-data.json`

```
{
  "penalties": [{
    "id": "uuid",
    "chatter_id": "uuid", "chatter_name": "Jane Doe",
    "offense_type": "Slow response time",
    "offense_number": 1,          // auto-increments per (chatter, type)
    "amount": 50,                 // USD, subtracted from payout
    "reason": "…full explanation…",
    "next_step": "…what happens if it repeats…",
    "week_key": "2026-06-15",     // Monday of the offense week
    "offense_date": "2026-06-21",
    "status": "active",           // active | void
    "acknowledged": true, "acknowledged_at": "ISO",
    "issued_by": "Admin",
    "created_at": "ISO"
  }],
  "types": ["Slow response time","Missed KPI","No-show","Late clock-in","
}
```

`availability-data.json`

```
{
  "submissions": [{
    "id": "…",
    "chatter_id": "uuid", "chatter_name": "Jane Doe",
    "week_key": "2026-07-06",     // the upcoming Monday
    "submitted_at": "ISO",
    "source": "grid",             // grid | admin | text_legacy
    "availability": { "mon": ["morning","evening"], "tue": ["night"], "…"
    "total_shifts": 5
  }]
}
```

`shift-corrections.json` **(pay disputes)**

```
{
  "corrections": [{
    "id": "…", "shift_id": 1234,
    "chatter_id": "uuid", "chatter_name": "Jane Doe",
    "shift_date": "2026-07-01", "shift_label": "…", "creator_name": "…",
    "original":  { "hours": 6, "shift_start": "…", "shift_end": "…", "tip
    "requested": { "note": "I actually worked 8h", "hours": 8, "shift_sta
    "status": "pending",          // pending | approved | rejected
    "submitted_at": "ISO", "reviewed_at": null, "reviewed_by": null, "rev
  }]
}
```

`chatter-groups.json` (creator bundles — the unit the builder schedules)

```
{
  "groups": [{
    "id": "…", "name": "Bundle A",
    "chatter_ids": [], "creator_ids": ["…","…"], "pinned_pairs": [],
    "created_at": "ISO", "updated_at": "ISO"
  }]
}
```

A **group** is a bundle of creators one chatter covers together in a single slot. This matters because you don't schedule chatter→creator one at a time; you schedule chatter→*group*, placing one person across the whole bundle in a single assignment.

Two time helpers you must get right

1. **ISO-week keying.** Key everything schedule-related to the **Monday** of its week. Write `cetWeekKey()` (Monday of the current week in your ops timezone), `nextWeekKey()` (upcoming Monday), and an ISO-week formatter (`Monday → "YYYY-Www"`). Consistency here saves you from an entire class of off-by-one bugs.

2. **The night-slot stored-date shift.** The night shift starts at 02:00. A shift someone works at 2am "Tuesday" belongs, operationally, to **Monday**

night. So when you store a night-slot row you shift its stored date back one calendar day (`_slotStoredDate(slot, displayDate)`: if the slot's start hour is < 10, subtract a day). Otherwise your coverage grid and your "who worked last night" queries will be wrong every single night. This one bites everybody. Handle it once, centrally.

Part 4 — The Weekly Schedule Builder (the centerpiece)

This is the part worth building. Here's the whole pipeline, admin-side.

Step 1 — Availability intake, with a hard deadline

`POST /api/chatters/availability` — a chatter submits a grid of `{ day: [slotIds] }` for the upcoming week. Validate against your real slot IDs and mon–sun keys. Key it to `nextWeekKey()`. Re-submitting overwrites.

The deadline is the whole trick. Availability **closes Friday 23:59** in your ops timezone. Submissions on Saturday/Sunday are rejected — "Deadline passed." Without a hard cutoff you can never build the schedule, because you're always waiting on the last two people. The deadline moves the pain from you to them. Admins get a bypass: `POST /api/admin/availability` lets you fill availability for someone (source `admin`) after the cutoff, so a legitimate straggler isn't a crisis — but the default is closed.

Step 2 — One call to open the builder

`GET /api/admin/schedule/build-context?week_key=YYYY-MM-DD` returns **everything the builder UI needs in one shot**: active chatters, approved holidays for that week, all creators (with their `assigned_chatters` allowlist and logos), that week's availability submissions, saved tier prefs, and the slot config. One request, whole picture. Don't make the UI stitch six calls together.

Step 3 — Tiers (weekly-shift caps)

You give each chatter a tier, saved via `PUT /api/admin/schedule/tier-prefs`. **The tier's only job is to cap weekly shifts** — think of it as a seniority/capacity band, not a ranking of who gets the good accounts. Tiers and their **weekly-shift caps**:

Tier	Meaning	Max shifts/week
S+	highest capacity	7
S		6
A		5
B		4
C	newest / part-time	3
X	exclude from auto-build	0

One job: **cap** — nobody burns out. Your S+ tops out at 7 shifts, your C at 3. X is the "don't auto-assign this person" escape hatch. The builder does **not** use the tier to decide who gets which creator — placement is driven by availability and fairness only (next step).

Step 4 — Auto-build: the availability bin-packing algorithm

`POST /api/admin/schedule/build-auto`. This is the deterministic scheduler. Here's exactly how it thinks:

1. **Take the creator groups in your configured order.** Give each group a simple priority number in the admin if you want a stable fill order (default them all equal). This order only breaks ties when bodies are scarce — it does not weight anyone toward "bigger" accounts.
2. **Eligibility filter.** A chatter is a candidate only if they (a) submitted availability, (b) are `active`, and (c) aren't tier X.
3. **Per-chatter weekly target** = `min(days they're actually available, TIER_MAX[tier])`. You never schedule someone more than they offered *or* more than their tier's cap allows, whichever is smaller.

4. **Iterate day → slot → group.** Critically, process the **scarcest slot first** — usually night, because fewest people want it. If you fill the easy slots first you strand the hard ones. Fill the hard ones while you still have bodies.
5. **Candidate test for a given day/slot/group:** available for that day+slot AND **not already assigned that day** (hard rule: **one shift per chatter per day**) AND still under their weekly target.
6. **Sort the candidates by this key, in order:**
    - **(1) Zero-shift floor** — anyone who submitted availability but has 0 shifts so far jumps the queue. Everyone who showed up to submit gets at least one shift. This is a morale rule as much as a math rule.
    - **(2) Furthest from their cap** — spread the load evenly; don't slam one person to their cap while someone else sits at 2.
    - **(3) Most days available** — flexibility gets rewarded with hours.
    - **(4) Stable tiebreak by id** — so the same input always produces the same schedule. Determinism matters; you want to be able to re-run and reason about it.
7. **Assign the winner to every creator in the group** — that's N schedule rows in one placement.
8. **Write everything as** `published: false` — it's a draft. Return `uncovered[]` (slots nobody could fill) plus a per-chatter `workload` / `tier_summary` so you can eyeball the result.

Then you clean up the gaps by hand: `POST /api/admin/schedule/cover-slot` fills an uncovered slot and deliberately **ignores** the availability and one-per-day rules (it's your manual override — sometimes you just need a body there).

There's also a plain writer, `POST /api/admin/schedule/build`, that takes an explicit `assignments[]` array with a dedup/idempotency guard, for when you want to place rows programmatically without the algorithm.

Step 5 — Manual editing tools

- `POST /api/admin/schedule` — add/replace one row.
- `POST /api/admin/schedule/bulk` — many at once.

- `DELETE /api/admin/schedule/:id` — remove one (cascade any orphaned dependent records).
- `GET /api/admin/schedule?week=` — read a week.
- `GET /api/admin/coverage?week=` — build the **7×3 coverage board** (days × slots) with a `covered` flag per cell. This is your at-a-glance "where are the holes" view.
- `POST /api/admin/schedule/copy-week` — clone one creator's week onto other creators, with `replace` / `skip` / `merge` modes. Enormous time-saver for stable rosters.

New rows inherit `published` only if that creator **already** has published rows for the week — publishing is per-creator, so editing one creator mid-week doesn't leak another creator's unpublished draft.

Step 6 — Publish (the draft→live gate)

Nothing reaches a chatter until you publish. `POST /api/admin/schedule/publish-week` (optionally scoped to `creator_ids`) or `/publish` (whole week) flips `published: true`. **Admin-only** — I explicitly 403 anyone who isn't a full admin (managers can build drafts but not publish). On publish, and *only* on publish, the system notifies each chatter in-app **and** DMs them their shifts on Telegram, localized to their timezone.

The draft→publish split is non-negotiable. You want to build, look at it, fix it, sleep on it, and *then* commit — without your team watching a half-built schedule flicker in and out and panicking.

Slot config lives at `GET/PUT /api/admin/slot-config`; a change fires a Telegram alert so everyone knows the shift windows moved.

Part 5 — The Penalties System

This is a proper conduct ledger, and it's separate from any per-shift dollar adjustment.

Issue a penalty

`POST /api/admin/penalties`. On issue it does four things:

1. Auto-computes `offense_number` — the Nth active offense of *this type* for *this chatter*. First slow-response is #1, next is #2. This is what makes escalation automatic and fair.
2. Pushes a persistent in-app notification (`type: 'penalty'`).
3. **DMs the chatter on Telegram** — "⚠️ Penalty issued … offense #N … open your portal → My Conduct to acknowledge."
4. Logs it to your activity feed with `issued_by` = whoever's logged in.

Manage

`GET /api/admin/penalties` (filter by chatter / type / week / status), `PUT /api/admin/penalties/:id` (editing the type re-numbers the offense), `DELETE`. Offense types are editable at `GET/PUT /api/admin/penalty-types` — start with: *Slow response time, Missed KPI, No-show, Late clock-in, Skipped mass messages, Other*.

How it hits pay — two separate mechanisms

This trips people up, so be precise:

- **(a) Ledger penalties** — conduct offenses. In the weekly payout roll-up, all active penalties whose `week_key` matches that week are summed and **subtracted as** `conduct_penalty`.
- **(b) Per-shift penalty** — the `shift.penalty` dollar field you set on one specific shift via `PUT /api/admin/shifts/:id`. Ad-hoc, tied to that shift, nets out separately.

Keep them separate. One is "your conduct this week cost you $X." The other is "this specific shift had a $Y adjustment." Conflating them makes payroll unauditable.

Real examples from the live ledger

These are real (names aside). Notice the tone — every penalty **explains** and every one has a **next step**. That's what turns a fine into a management tool.

- **Slow response — $50, offense #1.** *Reason:* "Your response time the last 2 weeks averaged 3m 47s, more than DOUBLE the standard. Rest of the team keeps close to 1m 30s, some below 1 minute." *Next step:* "Next slow-response period = fewer shifts + $100 penalty, then disqualification."
- **Slow response — $25.** *Reason:* "...averaged 2m 39s, over a minute above standard..." *Next step:* "Next period = fewer shifts + $75 penalty."
- **Slow response — $5 (warning tier).** *Reason:* "...averaged 1m 47s, 17s over standard. This is just a $5 warning, encouraging you below 1m 30s. You're very close, keep it up!" *Next step:* "$25 or more if it doesn't improve."
- **Slow response — $0.** A pure warning, no money, "4 seconds over, you're basically there."
- **Other / Breaking of TOS — $100.** *Next step:* "Immediate disqualification from the team."

See the graduated ladder? $0 → $5 → $25 → $50 → $100 → gone. The system enforces the ladder because the offense number is automatic. That's the whole value: **consistent, documented, escalating discipline that you don't have to remember to apply.**

Part 6 — Shift Tracking (clock-in → clock-out → payout)

Clock-in

`POST /api/chatters/clock-in`. Creates an `active` shift stamped with the current time, the slot the chatter picks, a fresh copy of the shift checklist, and `clocked_in_at`.

Two deliberate design decisions:

- **Guard:** if any active creator exists, the chatter *must* pick a creator on clock-in. This kills "null-creator" ghost shifts that you can never attribute or pay correctly.
- **No schedule gate.** Clock-in does **not** check the schedule. The schedule is *informational* — it tells people where they're expected, but the source of truth for pay is the clock. If someone covers an emergency shift they weren't scheduled for, they still clock in and still get paid. Decoupling `schedule[]` from `shifts[]` is what makes this work.

Late clock-in detection: if they clock in more than 15 minutes after slot start, fire a `late_clockin` alert to admins.

The shift checklist

Every shift carries a small task list the chatter ticks off (`PUT /api/chatters/shift-checklist/:itemId`). Mine is three items:

1. Send 3 Mass Messages.
2. Check the spender list + message the spender list (per SOP).
3. Check if a story is up — if not, post one.

Keep it short and non-negotiable. It's the "did you actually do the revenue-driving things" gate.

Clock-out and the payroll math

`POST /api/chatters/clock-out`. Compute `hours` from start/end (handle the midnight wrap for evening/night), then:

```
tips_net     = tips_gross     * 0.8      // platform takes 20%
messages_net = messages_gross * 0.8
total_net    = tips_net + messages_net
commission   = total_net * commission_rate   // e.g. 0.03
base_pay     = hours * hourly_pay
payout       = base_pay + commission
```

Set `status: 'completed'`, save the `handoff_note`. This math is the entire reason payroll arguments disappear — it's computed from timestamps and entered revenue, not from anyone's memory.

End-of-day (EOD) report — with a quality gate

`POST /api/chatters/eod`, keyed by shift, per creator: `total_sales, traffic_behaviour, key_conversations, turnover_info, other_concerns`.

Quality gate: if the three free-text notes total under ~130 characters, flag it (`eod_flagged_short`), DM the chatter to add detail, and alert admins. This one rule single-handedly kills the "traffic: good. sales: ok." garbage handoff. If the next chatter can't pick up where this one left off, the report failed.

The handoff feed shows the previous chatter's full EOD to whoever covers overlapping creators next — so context passes down the chain instead of resetting every shift.

Admin shift CRUD

`GET /api/admin/shifts` (join EOD data in), `POST` (manual add), `PUT /:id` (set bonus/penalty, mark paid → fires a payment notification, edit times/revenue, reassign creator), `DELETE /:id` (clean orphan EODs), `POST /:id/force-end`.

The watchdogs — in-memory cron sweeps (the safety net)

These are the difference between "I found out at 9am" and "I knew at 2:20am." Run them on `setInterval`.

- **Abandoned-shift sweep (every 5 min):** an active shift **2h past** its slot end → `stuck_chatter_alert` to admins. **3h past** → auto-complete it (`force_completed`, $0 revenue), fire an `abandoned_shift` emergency alert, and DM the chatter that it was auto-closed. This stops a forgotten shift from running forever and poisoning your hours math.
- **Missed-clock-in sweep (every 10 min):** a published, `assigned` schedule row for today that is 20 min–4h past slot start with **no matching shift record** → `missed_clockin` "No-Show" alert. This is your 20-minute early-warning on a dark shift.

- **Story-reminder sweep (every 10 min):** active shift on a creator with no story scheduled this slot → DM the chatter "No story scheduled — post one now."

Plus live admin widgets: `GET /api/admin/alerts` and `GET /api/admin/active-shifts` (story/mass-message coverage runway, overdue shifts, elapsed time, checklist progress) for the real-time dashboard.

Shift corrections (pay disputes, handled cleanly)

Chatter requests a change to hours/times/tips/messages on an **unpaid** shift (one pending request per shift). Admin approves — `POST /api/admin/shift-corrections/:id/approve` applies the change and **recomputes payout** — or rejects with a note. Now "I worked 8 not 6" is a structured request with an audit trail, not a DM fight.

Payouts

`GET /api/admin/payouts?week=` (per-chatter roll-up including bonus, per-shift penalty, and `conduct_penalty`), `GET /api/admin/payout-summary?week=` (P&L: `profit = revenue - payout`), `POST /api/admin/payouts/archive` (marks shifts paid + notifies). Weekly stats at `GET /api/admin/chatter-stats`.

Swap & coverage (chatter-initiated, admin-visible)

- **Call-out:** `POST /api/chatters/callout` marks the schedule row `open`, notifies all eligible non-holiday chatters, and sets a 2h escalation timer → `shift_uncovered` emergency alert if nobody claims it.
- **Claim:** `POST /api/chatters/claim-shift`.
- **Swaps:** request / accept / decline, with types `cover`, `slot_swap`, `cross_swap`.
- Admins see everything at `GET /api/admin/swap-requests`.

Part 7 — Notifications (Telegram): who gets pinged, and how

Don't build a notification center nobody checks. Push to Telegram. People already live there.

The five-bot split

Instead of one firehose bot, split notifications across **five typed bots** so people can tune what they get:

Bot	Emoji	Carries
shifts	🟢	routine shift lifecycle
emergency	🔴	no-shows, uncovered, abandoned
content	📦	content/story ops
team & money	👥	payments, registrations
automation	🤖	system/automation events

A registry maps each notification **type-string** to its owning bot, and a dispatcher routes each message. Admins/managers link the bots they want and set per-type on/off prefs. `notifyAdmins(type, text)` routes to the owning bot's subscribers (with a legacy fallback to a shared group). `notifyCMs(...)` is opt-in per type. Chatter DMs go through a single dedicated **chatter bot**, gated on that chatter having linked their `tg_chat_id` — no link, no DM.

Two nice touches: **coalescing buffers** (~90s) collapse a burst of per-creator clock-ins or EODs into one tidy message instead of ten pings, and every chatter DM is wrapped in horizontal rules so it reads clean on mobile.

Admins / managers receive

On the **shifts bot:** `shift_start`, `late_clockin`, `shift_end`, `eod_submit`, `eod_quality_flag`, `stuck_chatter_alert`, `schedule_published`,

`slot_config_updated`, `shift_callout`, `shift_claimed`, `swap_requested`, `swap_resolved`, `holiday_requested`, `holiday_changed`. On the **emergency bot:** `shift_uncovered`, `abandoned_shift`, `missed_clockin` (No-Show). On the **team & money bot:** payments, registrations.

The missed-clock-in / stuck / uncovered / auto-completed events are the shift-tracking safety net — those four are the ones that save you real money.

Chatters receive (on the chatter bot)

Schedule published (their shifts, timezone-localized), penalty issued, EOD receipt + EOD-too-short nudge, shift auto-closed, no-story reminder, and all the swap/coverage messages ("someone wants you to cover," "your shift was claimed," "swap accepted/declined"). Plus a persistent in-app notification list they can read and acknowledge.

Example message copy

- Admin, No-Show (emergency): `🔴 No-show: Jane was assigned Evening (18:00) on Creator X and hasn't clocked in — 22 min past start.`
- Chatter, penalty: `⚠️ Penalty issued: Slow response time (offense #2), $50. Open your portal → My Conduct to acknowledge.`
- Chatter, auto-close: `Your shift on Creator X was auto-closed after running 3h past its end. If that's wrong, submit a shift correction.`
- Admin, schedule published: `✅ Next week's schedule published. 41 shifts across 6 creators. 0 uncovered.`

Part 8 — Paste-ready build prompts for Claude Code

Open Claude Code in an empty folder and go top to bottom. Each prompt builds on the last. Read the output, run it, then move on — don't paste all six at once.

Prompt 1 — Skeleton + data layer

```
Build a Node/Express server (server.js) on port 3002 that serves a static
from the same folder and stores all state in flat JSON files. Create cach
load/save accessor pairs for these files, each returning sane empty defau
file is missing:
  - chatters-data.json  { chatters:[], shifts:[], schedule:[], holidays:{
      swap_requests:[], notifications:[], slot_config:{}, scheduler_tier_
  - penalties-data.json { penalties:[], types:[...] }
  - availability-data.json { submissions:[] }
  - shift-corrections.json { corrections:[] }
  - chatter-groups.json { groups:[] }
Add time helpers: cetWeekKey() and nextWeekKey() returning the Monday of
and next week in Europe/Berlin as YYYY-MM-DD; an ISO-week formatter Monda
and _slotStoredDate(slot, displayDate) that subtracts one day when the sl
hour is < 10 (night-slot fix). Seed slot_config with morning 10-18, eveni
night 02-10. Use the exact field shapes from the data-model spec I'll pas
```

Prompt 2 — Availability + the builder context

```
Add availability intake and the schedule builder context.
- POST /api/chatters/availability: accept { week_key, availability:{mon:[
  validate slot ids and mon-sun keys, key to nextWeekKey(), overwrite on
  HARD DEADLINE: reject after Friday 23:59 Europe/Berlin with "Deadline p
- POST /api/admin/availability: same but admin bypasses the deadline (sou
- GET /api/admin/schedule/build-context?week_key=: return in ONE response
  chatters, that week's approved holidays, all creators (with assigned_ch
  allowlist), that week's availability submissions, saved tier_prefs, and
- PUT /api/admin/schedule/tier-prefs: save per-chatter tier (S+,S,A,B,C,X
  is used ONLY as a weekly-shift cap, not as a ranking for who gets which
```

Prompt 3 — The availability bin-packing auto-builder

```
Implement POST /api/admin/schedule/build-auto. Deterministic algorithm:
  TIER_MAX = {S+:7,S:6,A:5,B:4,C:3,X:0}.   // caps only — tier does NOT r
  1. Take the creator groups in their configured order (optional per-grou
     number, default equal). Order only breaks ties when bodies are scarc
  2. Eligible chatter = submitted availability AND active AND tier ≠ X.
  3. weekly_target[chatter] = min(days available, TIER_MAX[tier]).
  4. Iterate day → slot (SCARCEST slot first, usually night) → group.
  5. Candidate = available that day+slot AND not already assigned that da
     (ONE shift per chatter per day) AND under weekly_target.
  6. Sort candidates by: (1) zero-shift-so-far first, (2) furthest from c
     (3) most days available, (4) stable by id.   // no tier/revenue weig
  7. Assign winner to EVERY creator in the group (N schedule rows), publi
     built_by_scheduler:'auto'.
  8. Return uncovered[] slots plus per-chatter workload/tier_summary.
Also add POST /api/admin/schedule/cover-slot to fill an uncovered slot ma
ignoring availability and the one-per-day rule.
```

Prompt 4 — Schedule editing, coverage board, publish

```
Add: POST /api/admin/schedule (single upsert), POST /api/admin/schedule/b
DELETE /api/admin/schedule/:id (cascade orphans), GET /api/admin/schedule
GET /api/admin/coverage?week= (build a 7x3 day-by-slot grid with a covere
POST /api/admin/schedule/copy-week (replace|skip|merge modes).
Publish workflow: POST /api/admin/schedule/publish-week ({week, creator_i
/publish (whole week). Set published:true. ADMIN ONLY (403 for managers).
inherit published only if that creator already has published rows for the
```

Prompt 5 — Shift tracking + payroll + watchdogs

```
Add the shift lifecycle:
- POST /api/chatters/clock-in: create an active shift at current CET time
  chosen slot + a fresh checklist copy; REQUIRE a creator if any active c
  NO schedule gate. Fire late_clockin if >15 min after slot start.
- PUT /api/chatters/shift-checklist/:itemId: toggle a checklist item.
- POST /api/chatters/clock-out: compute hours (handle midnight wrap), the
  tips_net=tips_gross*0.8, messages_net=messages_gross*0.8, total_net=sum
  commission=total_net*commission_rate, base_pay=hours*hourly_pay,
  payout=base_pay+commission; status='completed'; save handoff_note.
- POST /api/chatters/eod: per-creator notes; if the three notes total < 1
  flag eod_flagged_short, DM the chatter, alert admins.
- Admin: GET/POST/PUT/DELETE /api/admin/shifts(/:id) incl. mark-paid, bon
- GET /api/admin/payouts?week= (roll up bonus + per-shift penalty + condu
  GET /api/admin/payout-summary?week= (profit = revenue - payout).
Watchdogs on setInterval:
- every 5 min: active shift 2h past slot end → stuck_chatter_alert;
  3h past → auto-complete (force_completed, $0) + abandoned_shift alert
- every 10 min: published assigned row today, 20min-4h past start, no shi
- every 10 min: active shift, creator has no story this slot → DM story
```

Prompt 6 — Penalties ledger + Telegram notifications

```
Penalties:
- POST /api/admin/penalties: auto-set offense_number = Nth active offense
  for that chatter; push in-app notification; DM the chatter on Telegram;
- GET /api/admin/penalties (filter chatter/type/week/status), PUT /:id (r
  type changes), DELETE /:id, GET/PUT /api/admin/penalty-types.
- In payouts, subtract active penalties whose week_key matches as conduct
  (separate from the per-shift shift.penalty field).
Telegram: a bot registry of five typed bots (shifts, emergency, content,
mapping notification type-strings to bots; notifyAdmins(type,text) routes
bot's subscribers with per-type prefs; a single chatter bot gated on chat
~90s coalescing buffers for bursty clock-in/EOD messages. Wire every even
earlier steps to the right bot (shift_start/late_clockin/shift_end/eod_su
shifts; missed_clockin/abandoned_shift/shift_uncovered → emergency; paym
```

Part 9 — Notes, gotchas, and what you'll build on top

A few things to save you pain:

- **Security — do this right from day one.** This is an internal tool sitting behind an admin login, but it still handles your team's credentials and payout addresses, so don't cut corners on auth. **Hash passwords** with bcrypt or argon2 — never store them in plaintext. **Expire your session tokens** (and cap concurrent sessions) instead of letting them live forever. Put these in from the start on both the admin and chatter sides — retrofitting auth later is a pain, and this is the kind of thing you don't want to get wrong once real people are logging in.
- **Flat JSON is a feature until it isn't.** It's perfect up to a few thousand records and it makes your whole DB readable and version-controllable. If you get concurrent-write heavy or cross tens of thousands of rows, move the accessors to SQLite/Postgres — and because everything already goes through `lX()/sX()`, nothing else in the codebase changes.
- **This is a scaffold, not a product.** Claude Code will get you a running server, the algorithm, the routes, the data files, and the watchdogs. You still have to build the admin UI on top, test the edge cases against your real roster, and adapt slots/tiers/pay to your operation. Budget a weekend, not an afternoon.

What I'm deliberately NOT giving you: the chatter-panel wiring

Everything above is the **admin side and the system side** — the builder, the data model, the algorithm, the payroll math, the penalties ledger, the watchdogs, the notifications. That's the engine, and you can build the whole engine from this file.

The front-end you'll build on top for your own team is the **chatter-facing panel** — the login your chatters use to *see* their published schedule, read and acknowledge their penalties, clock in/out, tick their checklist, submit EOD and availability, dispute a shift, and manage swaps. That view — the

chatter login/session, the layout of their "here are my shifts / here are the open shifts" screen, the acknowledgement flow, the dispute UI — is where you make it your own, connected to the admin system you built above and styled how you want your team to use it.

Because **a schedule the team can't see is just a spreadsheet with extra steps** — the engine only pays off once your chatters see it, trust it, and act on it. So build the engine from this file (you've got everything for it here), then build the chatter panel on top and connect the two. That's the step that turns it into a system your team actually runs on.

You can build the whole engine this weekend with Claude Code. Go do it.
