Developer Guide

This guide explains how to work on Residency Rotation Scheduler (R2S), covering setup, architecture, key workflows, and day-to-day commands for both the FastAPI backend and the Vite/React frontend.

Repository layout

- `server/` : FastAPI app plus solver/pre/post-processing services. Entry point: `server/main.py` .
- `server/services/` : `preprocessing.py` (CSV ingest + validation), `posting_allocator.py` (OR-Tools model), `postprocess.py` (merge solver output, derive stats), `validate.py` (lightweight assignment checks for saves).
- `client/` : Vite + React + TypeScript UI with Tailwind. Entry point: `client/src/main.tsx` ; routing lives in `client/src/App.tsx` .
- `constraints.md` : Human-readable list of hard/soft constraints enforced by the solver.
- `README.md` : Product overview; this guide focuses on developer workflows.

Prerequisites

- Python 3.10+ with `pip` .
- Node.js 20+ with `npm` .
- OR-Tools depends on native binaries; run everything on x86/arm64 macOS or Linux. No extra system packages are required beyond Python headers.

Setup

From the repo root:

```
python -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install -r server/requirements.txt

cd client
npm install
```

Running locally

Start the API (from repo root):

```
uvicorn server.main:app --reload --port 8000
```

Start the client (separate terminal):

```
cd client
npm run dev -- --host --port 5173
```

The client expects the API at `http://localhost:8000` . Override with `API_BASE_URL` in a Vite env file (e.g., `client/.env.local`).

Data model and pipeline

1. **Upload/preprocess** (server/services/preprocessing.py)
   - Accepts multipart form uploads. Required CSVs: `residents` , `resident_history` , `resident_preferences` , `resident_sr_preferences` , `postings` ; optional `resident_leaves` . Columns must match `CSV_HEADER_SPECS` (aliases supported for camelCase flags). Parsing includes UTF-8 BOM stripping, strict header validation, duplicate checks (MCR, posting codes), integer parsing, and capacity/duration validation.
   - `weightages` (JSON string or dict), `pinned_mcrs` (JSON array), and `max_time_in_minutes` (positive int) are also read here.
2. **Solver** (server/services/posting_allocator.py)
   - Builds OR-Tools CP-SAT variables: block-level assignment (`x[mcr][posting][block]`), selection flags, run counts, and `off_or_leave` slack.
   - Normalises pins from current-year history + explicit `pinned_assignments` ; merges leaves (deduped by resident/block) and reserves capacity when a leave consumes a posting slot.
   - Derives stage progression per block (career blocks pause during leave) and historical SR-in-window markers.
   - Applies hard constraints listed in `constraints.md` (capacity, contiguity, CCR/GM/GRM/ED rules, MICU/RCCM packs, Dec→Jan guardrails, etc.) and scores soft goals (preference, seniority, elective/core shortfalls, bundle bonuses). Time limit uses `max_time_in_minutes` when provided.
3. **Postprocess** (server/services/postprocess.py)
   - Integrates solver entries with uploaded history to form `resident_history` (current year tagged), updates `career_blocks_completed` , and carries forward leave metadata.
   - Computes per-resident stats (core blocks, elective completions, CCR status, stages per block) and cohort stats (optimisation scores + normalised scores, preference satisfaction, posting utilisation).
4. **Save and CSV export**
   - `/api/save` validates user-edited schedules against a subset of hard rules (`server/services/validate.py`) before recomputing postprocess stats.
   - `/api/download-csv` returns `final_timetable.csv` with resident metadata, per-block postings, optimisation score, and CCR posting code.
5. **In-memory caching**
   - `server/main.py` keeps the latest inputs and API response in `Store` , enabling pinning and saves without re-uploading CSVs.

API surface

- `POST /api/solve` (multipart form): files named `residents` , `resident_history` , `resident_preferences` , `resident_sr_preferences` , `postings` , optional `resident_leaves` ; fields `weightages` (JSON), `pinned_mcrs` (JSON array of MCRs), `max_time_in_minutes` (int). Response contains `success` , `residents` , `resident_history` , `resident_preferences` , `resident_sr_preferences` , `postings` , `resident_leaves` , `weightages` , and `statistics` .
- `POST /api/save` (JSON): `{ resident_mcr: string, current_year: [{ month_block, posting_code }] }` . Uses cached dataset; returns the same shape as `/api/solve` on success.
- `POST /api/download-csv` (JSON): expects `success` , `residents` , `resident_history` , `optimisation_scores` ; returns a CSV blob.

Frontend architecture

- **API client**: `client/src/api/api.tsx` uses Axios; base URL = `import.meta.env.API_BASE_URL || http://127.0.0.1:8000/api`.
- **State**: `ApiResponseProvider` (`client/src/context/ApiResponseContext.tsx`) stores the latest API response in React context + `localStorage`.
- **Pages**: `DashboardPage` handles CSV uploads, weightages, solve trigger, pinning (persisted in `localStorage`), academic-year parsing, and renders resident-level timetables + cohort stats. `OverviewPage` is routed at `/overview`.
- **Components**: Key UI pieces under `client/src/components/` (file upload, weightage selector, resident timetable, statistics tables). `generateSampleCSV` can be triggered from the dashboard to download example datasets for quick smoke tests.
- **Styling**: Tailwind via `index.css`; layout composed in `Layout.tsx` with sidebar shell components under `components/ui`.

CSV schemas (quick reference)

- Residents: `mcr`, `name`, `resident_year`, `career_blocks_completed`.
- Resident History: `mcr`, `year`, `month_block` (1–12), `career_block`, `posting_code`, `is_current_year`, `is_leave`, `leave_type`.
- Resident Preferences: `mcr`, `preference_rank`, `posting_code`.
- SR Preferences: `mcr`, `preference_rank`, `base_posting`.
- Postings: `posting_code`, `posting_name`, `posting_type` (`core` / `elective`), `max_residents`, `required_block_duration`.
- Resident Leaves (optional): `mcr`, `month_block`, `leave_type`, `posting_code` (reserves capacity when set).

Development workflows

- **Lint/type-check frontend**: `cd client && npm run lint` and `npm run build` (runs `tsc -b` then Vite build).
- **Backend checks**: No automated tests provided; prefer running `uvicorn` locally with sample CSVs and checking logs. Add targeted unit tests under `server/` if you extend solver logic.
- **Logging**: Solver logs to stdout via `logging.basicConfig` in `posting_allocator.py`. Increase verbosity by adjusting the logger in that file while debugging.
- **Performance tips**: Keep CSVs small when iterating on constraints. Use `max_time_in_minutes` to cap long CP-SAT searches.

Common tasks

- **Change constraints/solver**: Modify `server/services/posting_allocator.py` and update `constraints.md` to keep docs aligned. Regenerate a timetable with sample CSVs to sanity-check feasibility and objective signals.
- **Adjust validation rules**: Update `server/services/preprocessing.py` (ingest-time) or `server/services/validate.py` (save-time) depending on where the guardrail should live.
- **Add new inputs**: Extend `CSV_HEADER_SPECS`, formatting helpers, and serialisation in preprocess; thread fields through solver payloads and postprocess outputs; update frontend types in `client/src/types.ts`.
- **API changes**: Keep `client/src/api/api.tsx` and context consumers in sync with backend response shapes.

Troubleshooting

- Infeasible solves: check backend logs for constraint violations, revisit pins/leaves/capacity, and review `constraints.md`.
- Upload failures: errors include the CSV label; ensure headers match the schemas above and files are UTF-8 encoded.
- Stale data after pinning or saving: the API caches the last run; restart the FastAPI process to clear state if needed.

Release/build pointers

- Frontend production build: `cd client && npm run build` (outputs `client/dist/`).
- Backend: run under a process manager (uvicorn/gunicorn) pointing at `server.main:app`; serve the built client separately (e.g., static host or reverse proxy) and set `API_BASE_URL` accordingly.

Production differences (Replit/hosted)

- **Static serving**: FastAPI serves `client/dist` directly; built client should use `VITE_API_BASE_URL=/api` to hit the same origin.
- **Stateless flow**: The in-memory `Store` is removed. `/api/save` requires the full prior `api_response` as `context`, and `/api/solve` supports pinned reruns via `previous_response`.
- **PostgreSQL sessions**: Set `DATABASE_URL` to enable auto-save on `/api/solve` and session CRUD routes (`/api/sessions`, `/api/sessions/latest`, `/api/db-status`, `/api/sessions/{id}`).
- **Auto-save**: Successful solves persist `api_response` to `solver_sessions`, returning `saved_session_id` when DB is available.
- **CORS/hosting**: CORS allows localhost and Replit domains; typical runtime command is `uvicorn server.main:app --host 0.0.0.0 --port ${PORT:-8000}` with the built client served by FastAPI. Ports 5000/5173 are used only for dev servers.