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. Developed and validated primarily on macOS (Apple Silicon); Linux should work with the same Python/Node versions and OR-Tools wheels.

Table of contents

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.

Branching model

- **dev**: default integration branch; cut feature branches from `dev` , open PRs back into `dev` , and keep it green (run lint/build smoke checks before merging).
- **prod**: protected release branch; merge only from `dev` during a release, then tag (e.g., `v1.0.0`). Avoid direct commits; fixes flow `feature -> dev -> prod` .

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`).

Local smoke test

1. Start the API and client as above.
2. In the dashboard, trigger the sample CSV generator (it downloads the CSV files) or use representative CSVs for residents, history (including leave rows), preferences (including SR base column), and postings.
3. Upload the CSVs, keep default weightages, and run the solver. Expect a successful timetable with optimisation scores populated; fix any schema errors shown in the UI or backend logs.
4. Pin a few assignments, re-run to confirm pins hold, and download `final_timetable.csv` to verify `/api/download-csv`.

Data model and pipeline

1. **Upload/preprocess** (server/services/preprocessing.py)
   - Accepts multipart form uploads. Required CSVs: `residents`, `resident_history`, `resident_preferences`, `postings`. 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

Can be viewed at http://127.0.0.1:8000/docs

- `POST /api/solve` (multipart form): files named `residents` , `resident_history` , `resident_preferences` , `postings` ; 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` , `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.

Tech Stack

- **Frontend**: React, Vite, Typescript
- **Backend**: FastAPI, Google OR-Tools, Python
    - Reference: Scheduling
- **Deployment**: Replit

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`.

Input 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` (use `is_leave=1` rows for leave entries).
- Preferences: `mcr`, `preference_rank`, `posting_code`, `resident_sr_preferences` (optional per row).
- Postings: `posting_code`, `posting_type` (`core` / `elective`), `max_residents`, `required_block_duration`.

Data Structure of helpers in `posting_allocator.py`

Refer to `# HELPERS` section of the code in server/services/posting_allocator.py.

`posting_info`

- Map of posting codes to posting info
- **Type**: `Dict[str, Dict[str,Any]]`
- **Shape**:

```
posting_info[posting_code] = {
  "posting_code": str,
  "posting_type": str,
  "max_residents": int,
  "required_block_duration": int
}
```

- **Example**:

```
posting_info = {
  "GM (TTSH)": {
      "posting_code": "GM (TTSH)",
      "posting_type": "core",
      "max_residents": 5,
      "required_block_duration": 1
  },
  ...
}
```

`posting_codes`

- List of posting codes
- **Type**: `List[str]`
- **Example**:

```
posting_codes = [
  ED (TTSH),
  Endocrine (TTSH),
  Gastro (TTSH),
  GM (TTSH),
  GRM (TTSH),
  ...
]
```

`CORE_POSTINGS`

- Filtered list of `posting_codes`, of core postings
- **Type**: `List[str]`
- **Example**:

```
CORE_POSTINGS = [
  CVM (TTSH),
  ED (TTSH),
  GM (TTSH),
  GRM (TTSH),
  ...
]
```

`ELECTIVE_POSTINGS`

- Filtered list of `posting_codes`, of elective postings
- **Type**: `List[str]`
- **Example**:

```
ELECTIVE_POSTINGS = [
  ID (TTSH),
  Med Onco (TTSH),
  PMD (TTSH),
  RAI (TTSH)
  ...
]
```

`ELECTIVE_BASE_CODES`

- Unique list of elective posting codes without variants (without institution names)
- **Type**: `List[str]`
- **Example**:

```
ELECTIVE_POSTINGS = [
  ID,
  Med Onco,
  PMD,
```

```
  RAI
  ...
]
```

`pref_map`

- Map of resident `mcr` to their elective preferences
- **Type**: `Dict[str, Dict[int,str]]`
- **Shape**:

```
pref_map[mcr] = {
  "preference_rank": "posting_code"
}
```

- **Example**:

```
pref_map = {
  "M000001A": {
      1: "Renal (TTSH)",
      2: "Endocrine (TTSH)",
      3: "Gastro (TTSH)",
      ...
  },
  ...
}
```

`sr_pref_map`

- Map of resident `mcr` to their SR preferences (base postings)
- **Type**: `Dict[str, Dict[int,str]]`
- **Shape**:

```
sr_pref_map[mcr] = {
  "preference_rank": "base_posting"
}
```

- **Example**:

```
pref_map = {
  "M000001A": {
      1: "Renal",
      2: "Endocrine",
      3: "Gastro",
      ...
  },
  ...
}
```

`pins_by_resident`

- Map of resident `mcr` to the block and posting code
- **Type**: `Dict[str, Dict[int,str]]`
- **Shape**:

```
pins_by_resident[mcr] = {
  "block": "posting_code"
}
```

- **Example**:

```
pins_by_resident = {
  "M000001A": {
      6: "ED (TTSH)",
      7: "GM (TTSH)",
      9: "Gastro (TTSH)",
      ...
  },
  ...
}
```

`resident_history`

- List of resident history (= `filtered_resident_history`)
- **Type**: `List[Dict]`
- **Shape**:

```
resident_history = [
  {
      "mcr": str,
      "year": int
      "month_block": int,
      "career_block": int,
      "posting_code": str,
      "is_current_year": int,
      "is_leave": int,
      "leave_type": str,
  }
]
```

- **Example**:

```
resident_history = [
  {
    "mcr": "M000001A",
    "year": 2,
    "month_block": 3,
    "career_block": 19,
    "posting_code": "GM (TTSH)",
    "is_current_year": 0,
    "is_leave": 0,
    "leave_type": "LOA",
  },
  ...
]
```

`resident_leaves` (= `normalised_leaves`)

- List of leave info derived from current-year `resident_history` rows with `is_leave=1` (same structure as `derived_leave_rows`).
- **Type**: `List[Dict[str,Any]]`
- **Shape**:

```
derived_leave_rows = [
  {
    "mcr": str,
    "month_block": int,
    "posting_code": str,
    "leave_type": str,
  }
]
```

- **Example**:

```
derived_leave_rows = [
  {
    "mcr": "M000001A",
    "month_block": 3,
    "posting_code": "GM (TTSH)",
    "leave_type": "LOA",
  },
  ...
]
```

`posting_progress`

- Map of each resident to their posting code and posting progress
- **Type**: `Dict[str, Dict[str, Dict[str, int]]]`
- **Shape**:

```
posting_progress[mcr] = {
    "posting_code": {
      "blocks_completed": int,
      "blocks_required": int,
      "is_completed": bool
    }
}
```

- **Example**:

```
posting_progress = {
  "M000003A": {
    "Renal (TTSH)": {
      "blocks_completed": 1,
      "blocks_required": 2,
      "is_completed": 0
    },
    ...
  },
```

```
  ...
}
```

`leave_off_blocks`

- List of unique tuples of `mcr` and leave block
- **Type**: `Set[Tuple[str, int]]`
- **Shape**:

```
leave_off_blocks = {
  (
    "mcr": str,
    "block": int
  )
}
```

- **Example**:

```
leave_off_blocks = {
  (
    "mcr": "M000001A",
    "block": "3"
  ),
  ...
}
```

`leave_map`

- Map of `mcr` to month block with leave type and posting code
- **Type**: `Dict[str, Dict[int, Dict[str,str]]]`
- **Shape**:

```
leave_map[mcr] = {
  block: {
    "leave_type": str,
    "posting_code": str
  }
}
```

- **Example**:

```
leave_map = {
  "M000001A": {
    4 : {
      "leave_type": "LOA",
      "posting_code": "Renal (TTSH)"
    }
  },
  ...
}
```

`leave_quota_usage`

- Tracks how many posting slots are consumed by leave, per posting and per block
- **Type**: `Dict[str, Dict[int, int]]`
- **Shape**:

```
leave_quota_usage[posting_code] = {
  "block": int
}
```

- `leave_quota_usage[posting_code][block]` = Integer count of leave entries reserving that posting in that block
- **Example**:

```
leave_quota_usage = {
  "GM (TTSH)": {
    4 : 2
  },
  ...
}
```

`career_progress`

- **Type**: `Dict[str, Dict[str, Any]]`
- **Shape**:

```
career_progress[mcr] = {
  "completed_blocks": int,
  "stage": int,
  "stages_by_block": Dict[int, int],
  "career_blocks_by_block": Dict[int, Optional[int]]
}
```

- **Example**:

```
career_progress = {
  "M000001A": {
    "completed_blocks": 20,
    "stage": 2,
    "stages_by_block": {
      1 : 2,
      2 : 2,
      3 : 2,
      4 : 3,
      ...
    }
    "career_blocks_by_block": {
      1 : 22,
      2 : 23,
      3 : None,
      4 : 24,
      ...
    }
```

```
  }
}
```

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.

API error responses

- **200 OK**
  - `/api/solve`
    - Response: `{ "success": true, "residents": [...], "resident_history": [...], "statistics": {...}, ... }` (per-resident `warnings` are present but currently returned as empty lists).
    - Reason: Solve completed successfully.
  - `/api/save`
    - Response: same shape as `/api/solve`.
    - Reason: Save succeeded and timetable recomputed.
  - `/api/download-csv`
    - Response: streamed `final_timetable.csv` with `Content-Disposition: attachment`.
    - Reason: Provided payload already matches solver response shape.

- **400 Bad Request**
  - `/api/solve`
    - Response: `{"detail": "...reason..."}`.
    - Reason: Input validation failed. Examples: missing uploads (`"Missing required file 'residents'. Received type 'None'. Available keys: [...]"`), CSV header/format issues (`"[Residents CSV] Missing required column(s): mcr, name."`, `"[Resident History CSV] Invalid month_block '13' for resident ABC: must be between 1 and 12."`), duplicate keys (`"[Postings CSV] Duplicate posting_code(s) found: XYZ. Each posting must have a unique code."`), invalid capacity/duration, UTF-8 decode failures, or pinning without cache (`"No existing timetable found. Upload CSV files before pinning residents."`).
  - `/api/save`
    - Response: `{"detail": "missing resident_mcr"}` when MCR is blank; `{"detail": "No dataset loaded. Upload CSV and run optimiser first."}` when nothing is cached; or validation payload `{ "success": false, "warnings": [ { "code": "HC3", "description": "..." }, ... ] }`.
    - Reason: Required fields missing or validation of the proposed schedule failed (constraint codes: INPUT, HC3, HC4, HC5, HC6, HC7a, HC7b, HC8, HC9, HC10, HC11, HC12).
  - `/api/download-csv`
    - Response: `{"detail": "Invalid API response shape"}`.
    - Reason: Payload did not include valid `success`, `residents`, `resident_history`, or `optimisation_scores` fields.
- **500 Internal Server Error**
  - `/api/solve`
    - Response: `{"detail": "Solver could not find a solution."}` or `{"detail": "Solver reported the model as invalid."}` or `{"detail": "Posting allocator service failed unexpectedly."}` or `{"detail": "Postprocess failed"}` (or other postprocess error string) or `{"detail": "Failed to process files"}`.
    - Reason: Optimiser/postprocess crashed or returned failure after inputs were accepted.
  - `/api/save`
    - Response: `{"detail": "Postprocess failed"}` (or a more specific postprocess error string).
    - Reason: Recomputing the timetable after save succeeded in parsing but failed during postprocessing.

Release checklist

- Update `CHANGELOG.md` and version mentions (this guide and `README.md`).
- Run `cd client && npm run lint && npm run build`; ensure the API accepts a representative dataset without errors.
- Perform the local smoke test above and download `final_timetable.csv` as a sanity check.
- For hosted runs, build the client (`npm run build`), set `VITE_API_BASE_URL` / `API_BASE_URL`, and confirm DB-backed session endpoints if `DATABASE_URL` is set.

- Tag the release (e.g., `v1.0.0`) and capture links to any sample datasets used for verification.

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.