Codoc in Vecdraw Project Design & Architecture Overview

1) Project Positioning and Core Concepts

- This is a **collaborative vector drawing editor** built with **Reflex 0.8.23** (similar to Google Drawings).
- It uses **EditorState (rx.SharedState)** as the “multi-user shared” state core.
- The UI is composed of several key components: Toolbar / Topbar / Canvas / Properties Panel.
- It supports:

- Basic shapes: rectangle, ellipse, triangle, line, text, pencil (path), image (upload)

- Operations: select, drag, resize, pan (hand tool), undo/redo, delete
- Export: SVG / PNG / JSON (via `assets/export_canvas.js`)
- AI/MCP operations: an external MCP Server sends ops -> Reflex receives them -> the frontend polls and triggers application

—

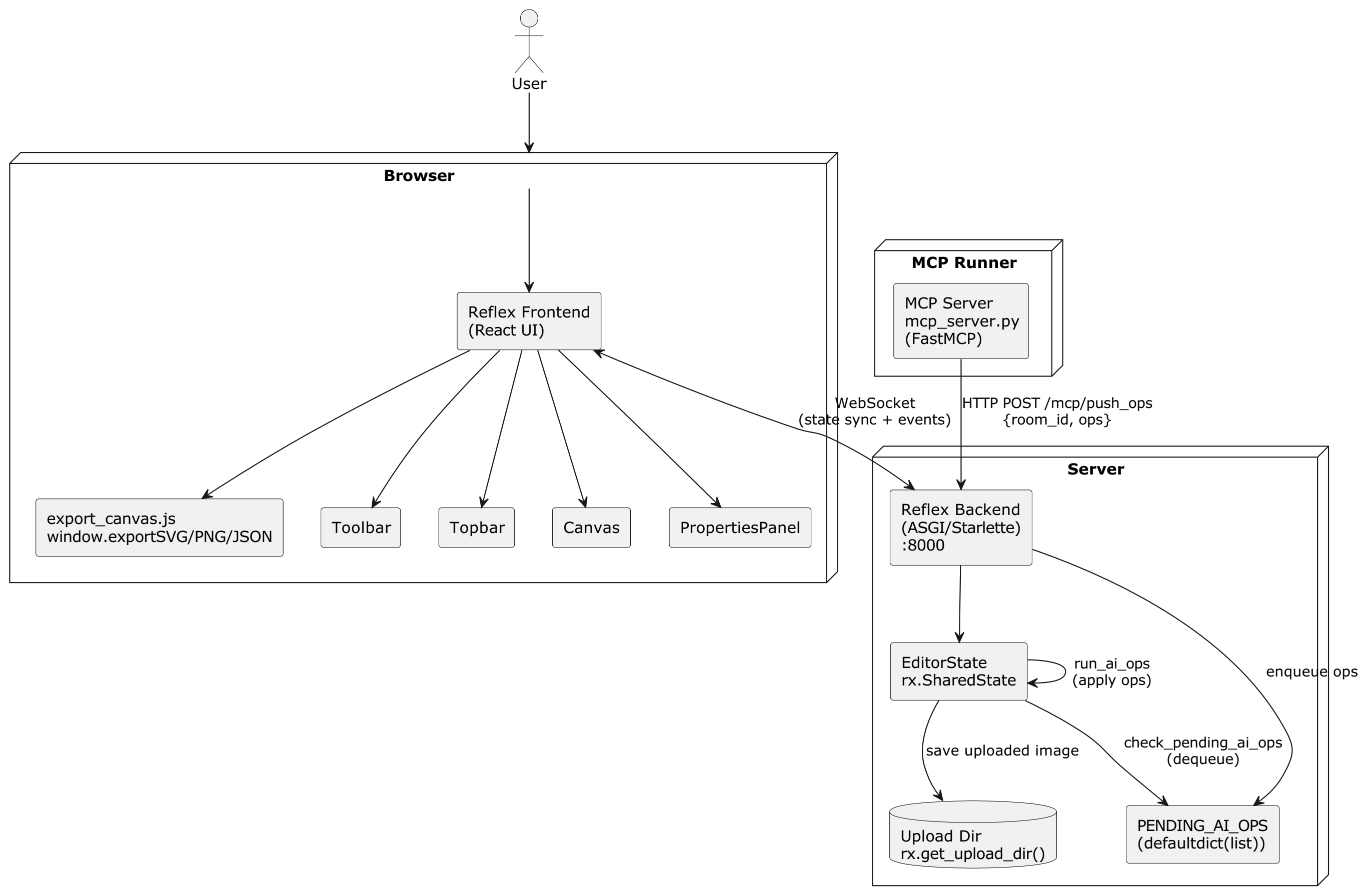
2) Repo Map (Key Files)

```
codoc_in_vecdraw-main/
  assets/
    export_canvas.js          # window.exportSVG/exportPNG/exportJSON
  codoc_in_vecdraw/
    codoc_in_vecdraw.py       # Reflex page + /mcp/push_ops API route
    components/
      toolbar.py              # Toolbar (Select/Hand/Shape/Text/Pencil/Upload)
      topbar.py               # Undo/Redo/Export/Room/AI modal
      canvas.py               # SVG canvas + mouse events
      shapes.py               # render_shape/render_preview
      properties_panel.py     # Right-side property editor + delete
    states/
      editor_state.py         # EditorState(rx.SharedState) +
                              # Shape model + ops/undo/redo
  mcp_server.py               # FastMCP tools -> POST /mcp/push_ops
  rxconfig.py                 # Reflex config
  testcases/                  # Playwright E2E (debug_shapes/debug_text/debug_image)
  run_test_suite.sh           # Start app + run tests
```

—

3) Architecture Overview (Component Diagram)

> This diagram connects the Reflex frontend/backend, SharedState, multiple UI components, MCP, and the export JS.

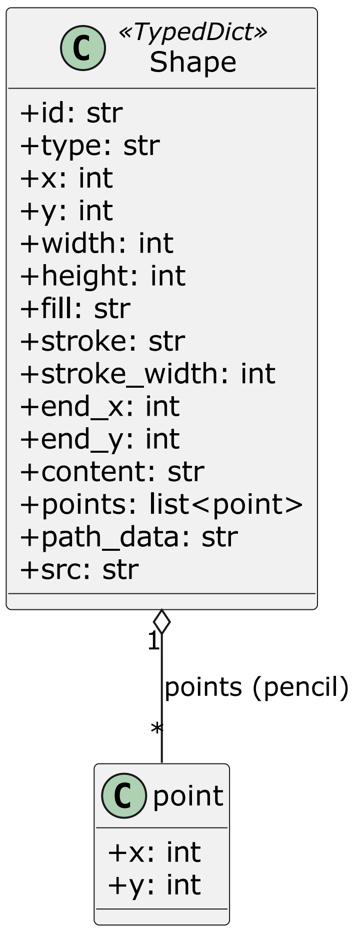

User
Browser
Reflex Frontend (React UI)
export_canvas.js window.exportSVG/PNG/JSON
Toolbar
Topbar
Canvas
PropertiesPanel
MCP Runner
MCP Server mcp_server.py (FastMCP)
WebSocket (state sync + events)
HTTP POST /mcp/push_ops {room_id, ops}
Server
Reflex Backend (ASGI/Starlette) :8000
EditorState rx.SharedState
run_ai_ops (apply ops)
enqueue ops
save uploaded image
check_pending_ai_ops (dequeue)
Upload Dir rx.get_upload_dir()
PENDING_AI_OPS (defaultdict(list))

—

4) Core Data Models (Shape / EditorState)

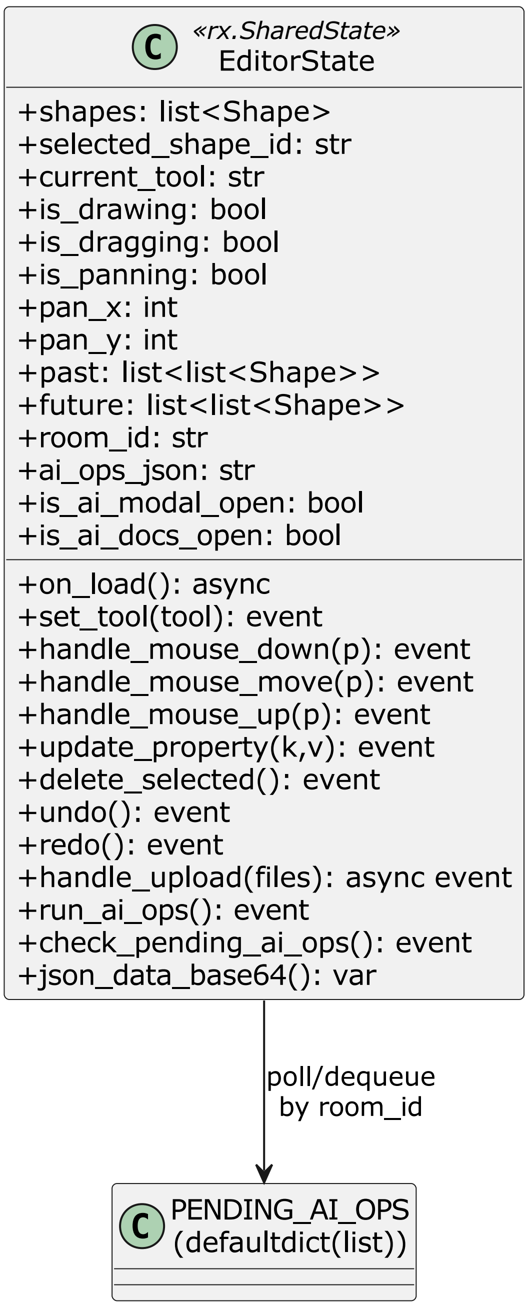
4.1 Shape (TypedDict) Field Concept

Each shape is a `Shape` dict (a single structure that can represent multiple types).



4.2 EditorState (SharedState) as the Collaboration Core

- `EditorState` inherits **rx.SharedState**, meaning multiple users in the same token/room share the same state.
- `on_load()` reads the query `?room=...` and calls `_link_to(token)` to connect the user to the same shared state.



—

5) UI Composition and Responsibilities

5.1 Toolbar (Left Tool Panel)

- `Select`, `Hand (Pan)`, `Rectangle`, `Ellipse`, `Triangle`, `Line`, `Pencil`, `Text`
- Image Upload: calls `EditorState.handle_upload(files)`, writes files into the upload dir, then adds a `type="image"` shape.

5.2 Canvas (Main Drawing Area)

- Uses `rx.call_script(GET_COORDS_SCRIPT, callback=...)` to get mouse coordinates.
- Mouse event mapping:

- down -> `handle_mouse_down`
- move -> `handle_mouse_move`
- up/leave -> `handle_mouse_up`
- Applies viewport translation using `EditorState.pan_x/pan_y`.
- Enters different interaction modes based on `EditorState.current_tool` (drawing / panning / selecting / dragging / resizing).

5.3 Properties Panel (Right-side Properties)

- Shows properties of the selected shape (fill/stroke/stroke_width/content, etc.)
- Directly calls `update_property`
- Provides `delete_selected`

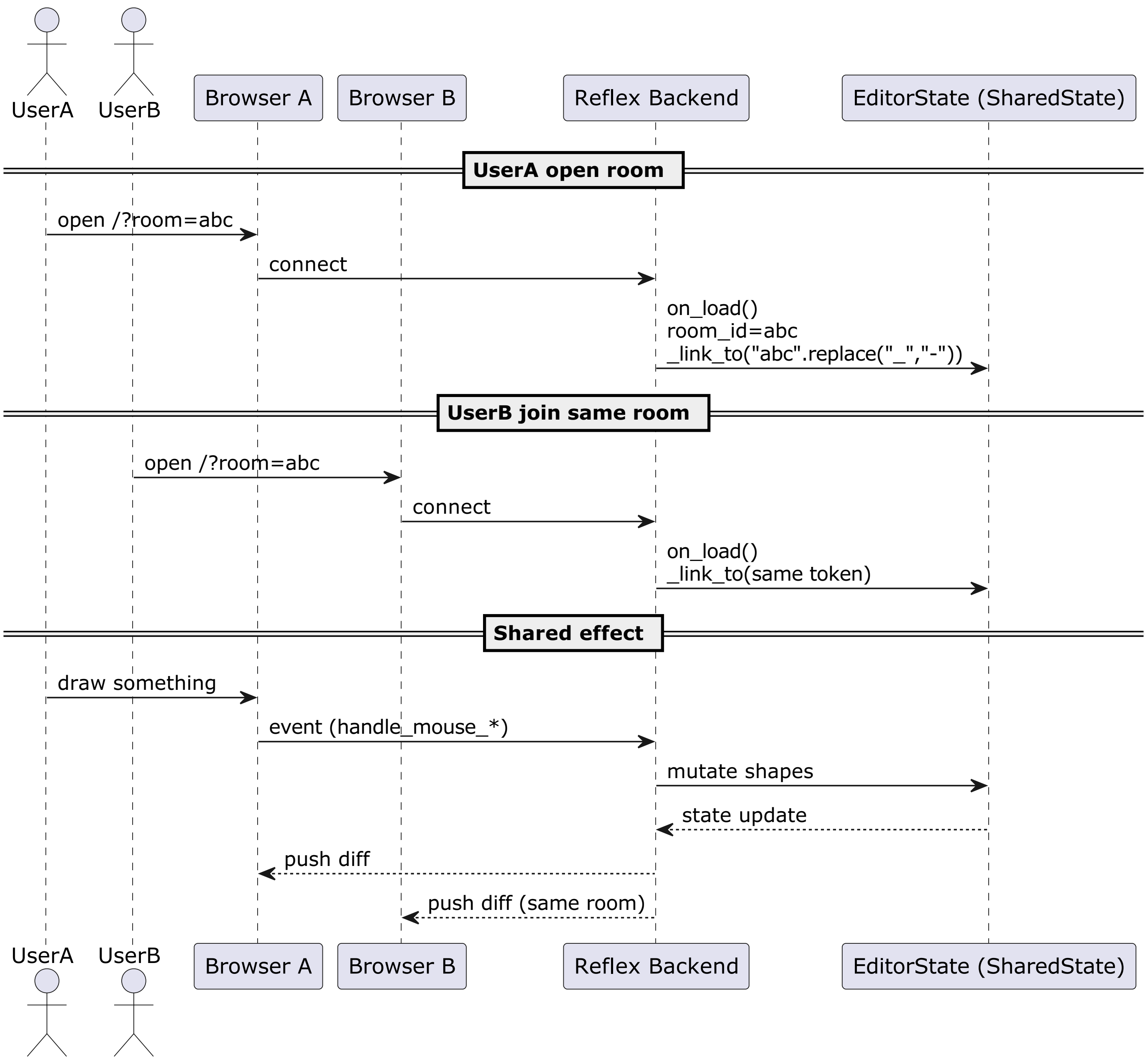
5.4 Topbar (Top Bar)

- Undo/Redo
- Room create/copy link
- Export (JSON/SVG/PNG)
- AI modal: paste ops JSON or receive ops via MCP

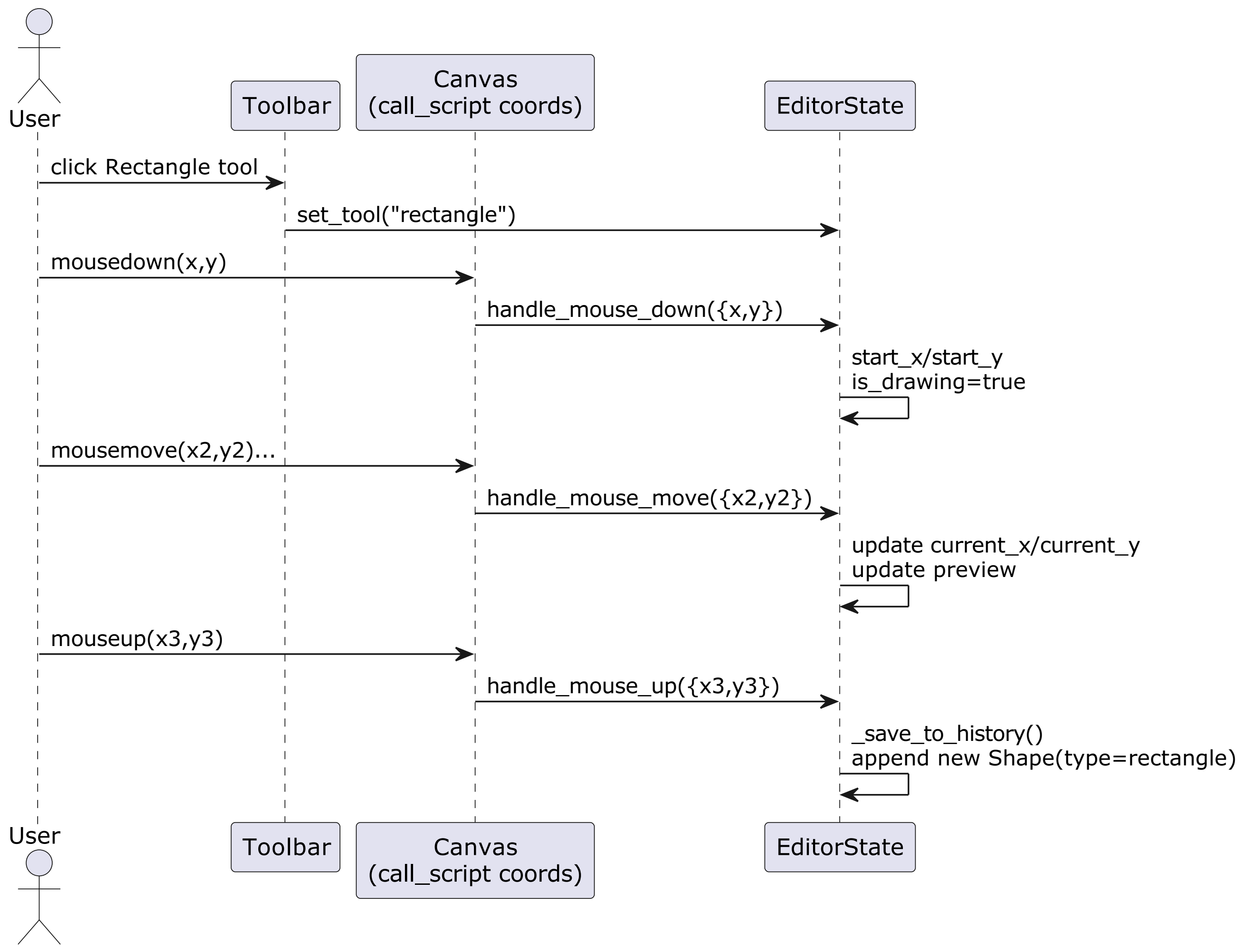
—

6) Key Interaction Flows (Sequence Diagrams)

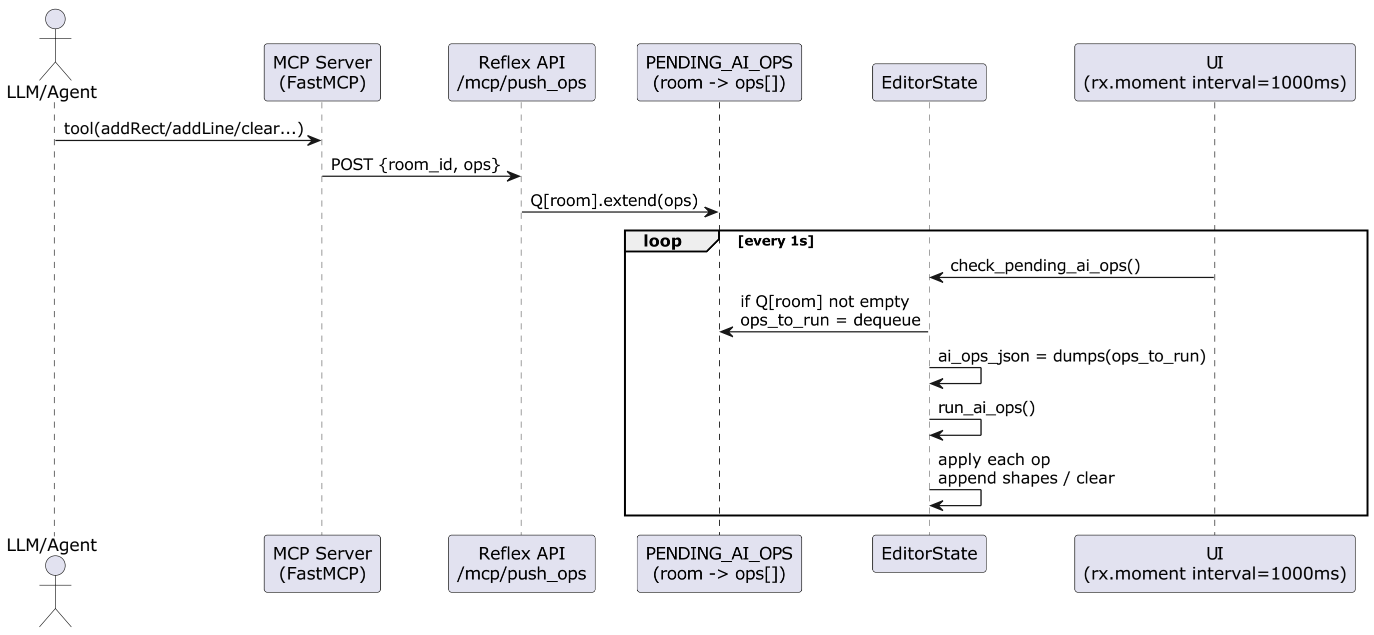
6.1 Collaboration Connection Flow for the Same Room (SharedState)



6.2 Drawing a Rectangle on the Canvas



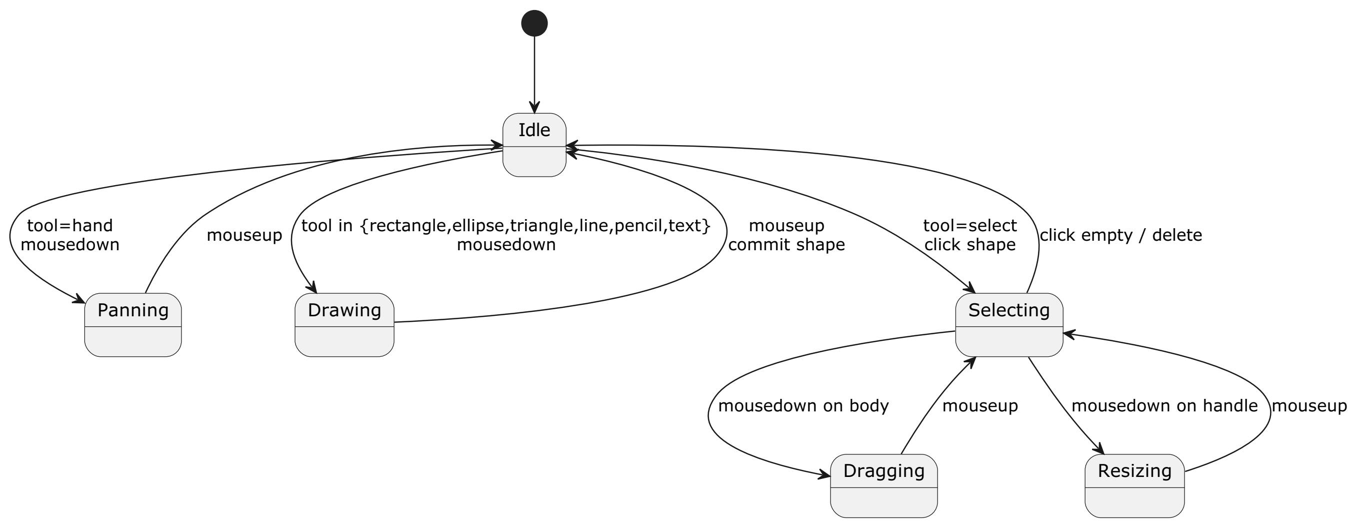
6.3 External MCP Control (ops -> /mcp/push_ops -> queue -> poll -> apply)



—

7) Interaction State Machine (State Diagram)

> From the user’s perspective: how Select/Resize/Drag, Hand(Pan), and drawing tools switch inside `EditorState`.

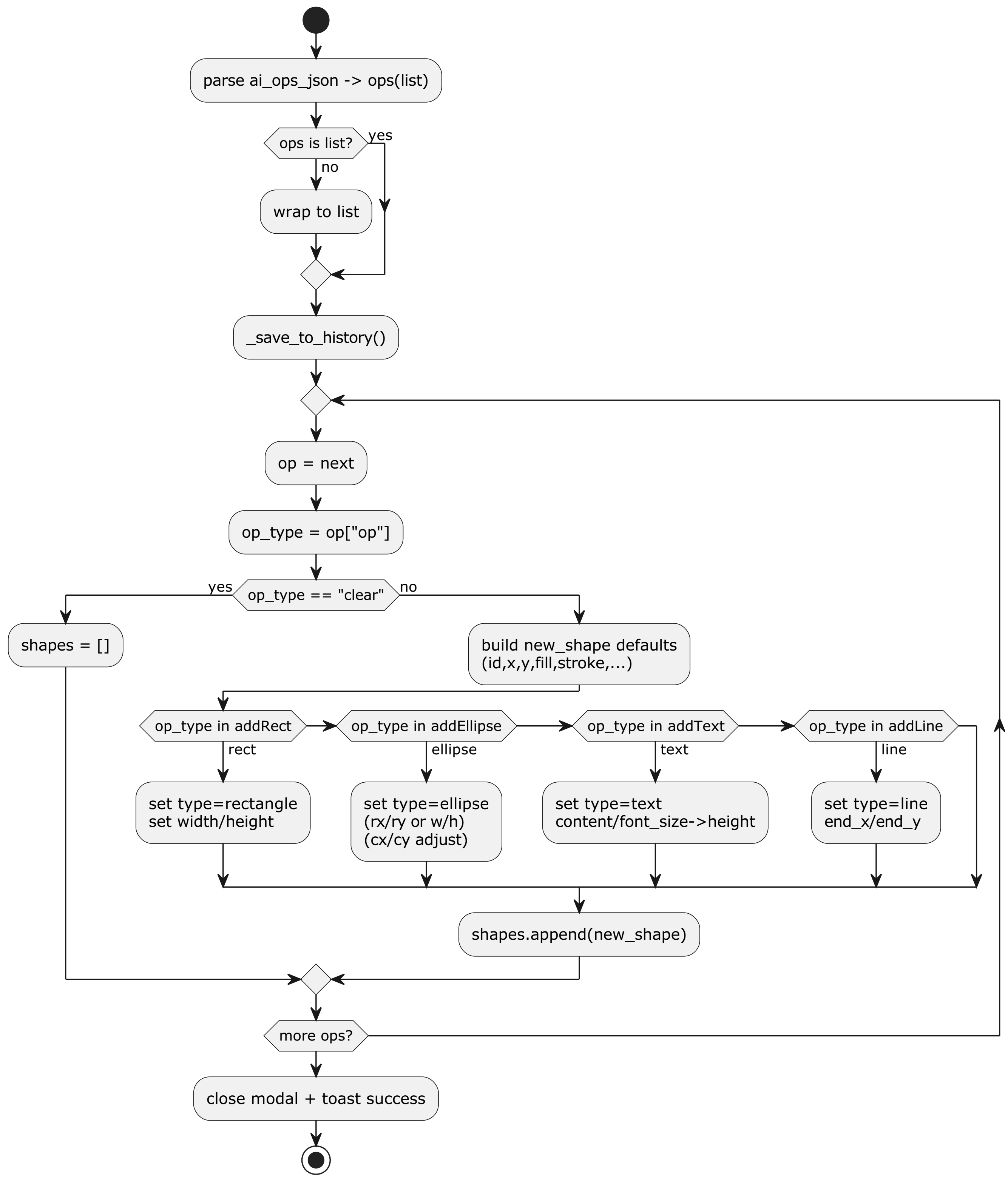


—

8) AI Ops Execution Logic (Activity Diagram)

Currently supported ops in `run_ai_ops()` (based on the implementation):

- `clear`
- `addRect` / `add_rectangle`
- `addEllipse` / `add_ellipse` (supports `rx/ry` or `width/height`, and supports converting `cx/cy` to top-left)
- `addText` / `add_text`
- `addLine` / `add_line`


parse ai_ops_json -> ops(list)
ops is list?
yes
no
wrap to list
_save_to_history()
op = next
op_type = op["op"]
op_type == "clear"
yes
no
shapes = []
build new_shape defaults (id,x,y,fill,stroke,...)
op_type in addRect
rect
set type=rectangle set width/height
op_type in addEllipse
ellipse
set type=ellipse (rx/ry or w/h) (cx/cy adjust)
op_type in addText
text
set type=text content/font_size->height
op_type in addLine
line
set type=line end_x/end_y
shapes.append(new_shape)
more ops?
close modal + toast success

—

9) Export Design

- `assets/export_canvas.js` registers browser-side functions:
- `window.exportSVG()`
- `window.exportPNG()`

- `window.exportJSON(jsonString)`
- The Topbar Export Menu calls:

- JSON: base64-encodes the shapes JSON via `EditorState.json_data_base64`, then passes it to `window.exportJSON(atob(...))`

- SVG/PNG: directly calls the corresponding window function

You can view export as “frontend reads the SVG in the DOM -> serializes -> downloads”, while JSON is “state -> base64 -> frontend decodes -> downloads”.

—

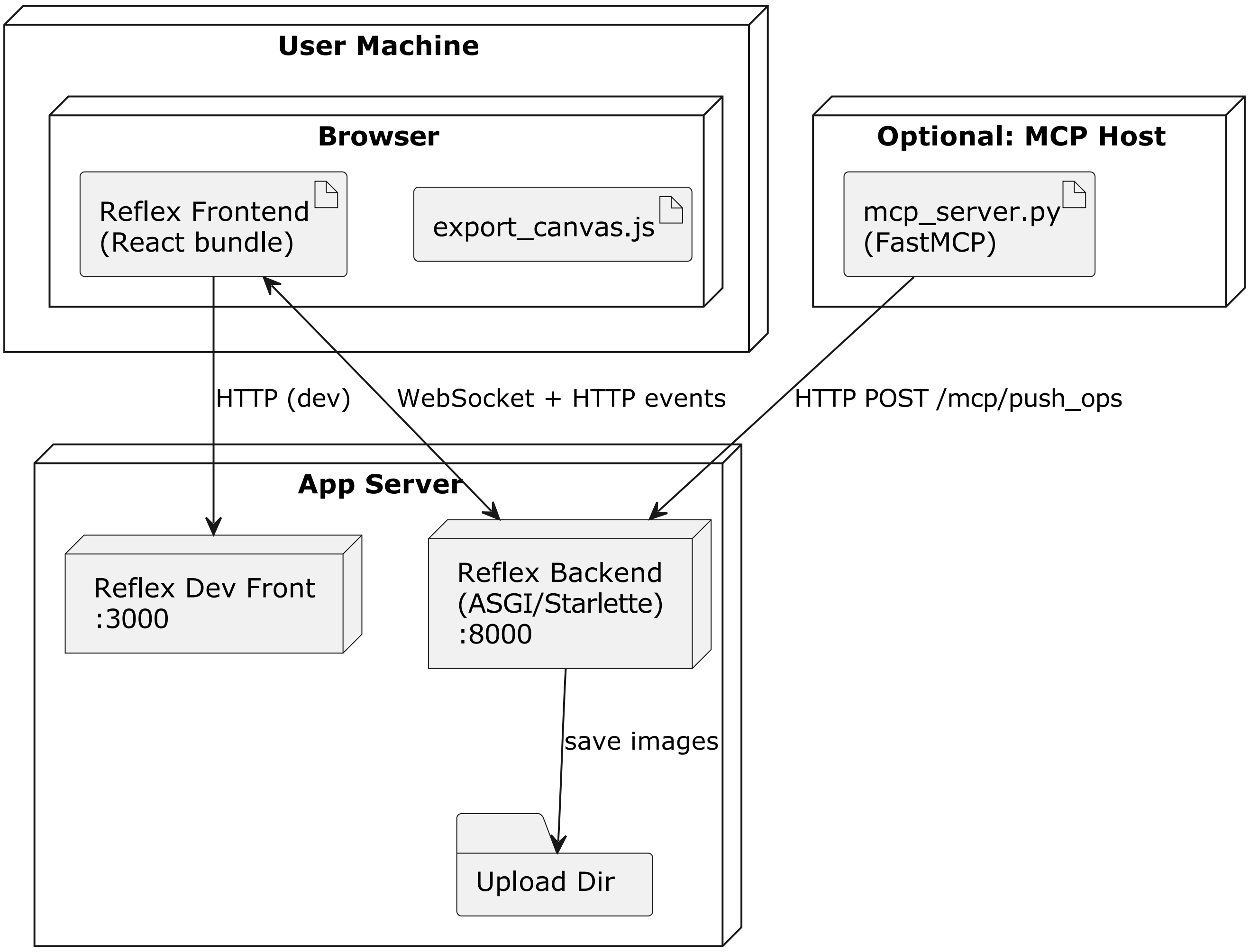
10) Testing (Playwright E2E) and Execution Script

- `testcases/*/run_test.py`: uses Playwright to drive browser actions and take screenshots (debug_shapes/debug_text/debug_image)
- `run_test_suite.sh`: roughly:

1. Kill old processes
2. Start the Reflex app
3. Run the specified test cases

—

11) Deployment View (Deployment Diagram)



—

Codoc in Vecdraw Project Design & Architecture (Details)

A) Collaboration room/token: From Room Creation to Real-time Multi-user Sync

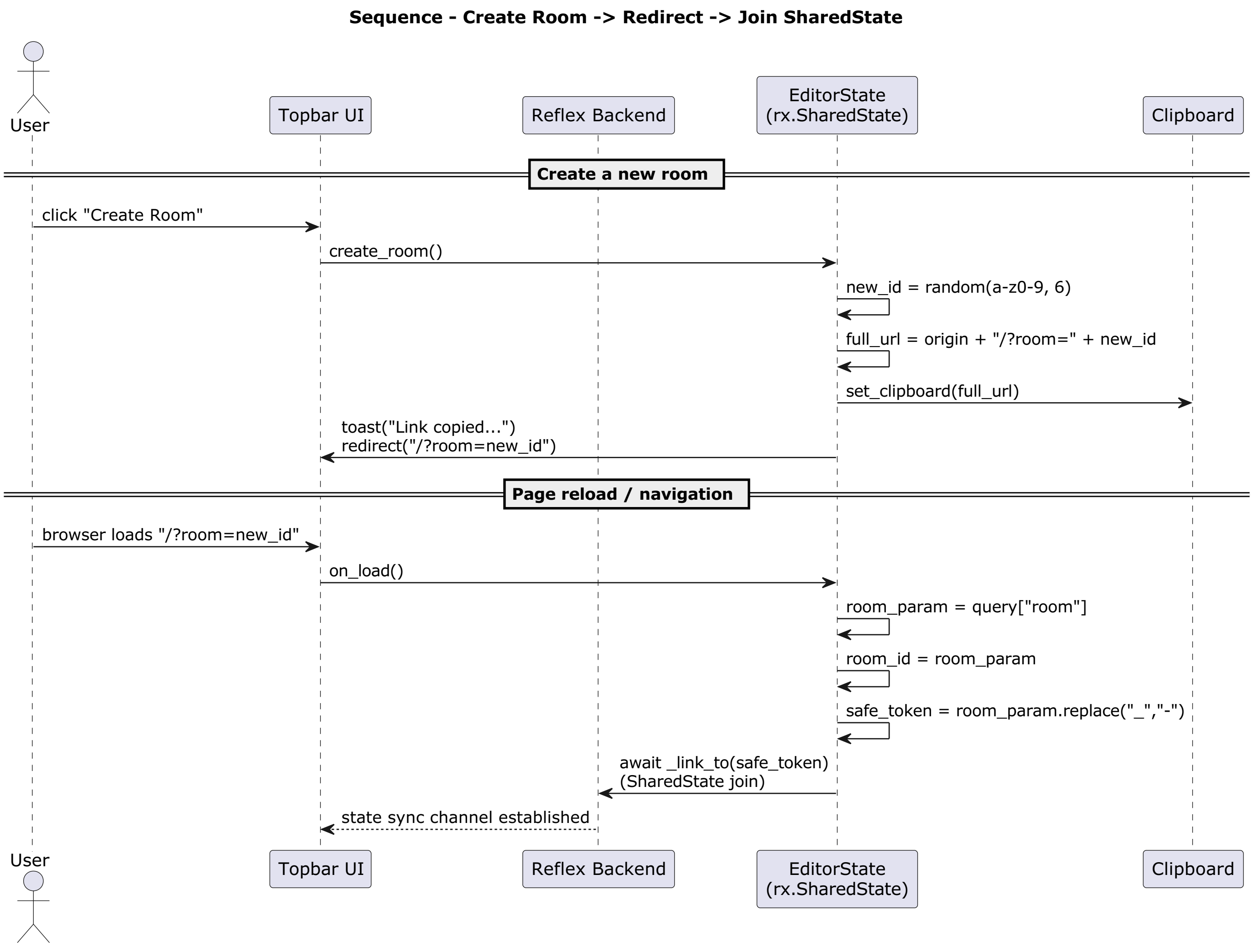
A1. Room Generation and Sharing (create_room / copy_room_link)

- `EditorState.create_room()`:
- Generates `new_id`: `[a-z0-9]` with length `6`
- Builds the URL: `{origin}/?room={new_id}`
- Copies the link via `rx.set_clipboard(full_url)`

- Redirects into the room via `rx.redirect("/?room=new_id")`
- `EditorState.on_load()`:
- Reads the query `?room=...`
- `self.room_id = room_param`
- `safe_token = room_param.replace("_","-")`
- `await self._link_to(safe_token)`: links the current connection to the same SharedState token (the key for shared multi-user state)

> Note: The room id itself does not actually generate `_`, but the code still replaces it to avoid unsafe characters in the token.

Sequence: Create Room -> Copy Link -> Redirect -> Join

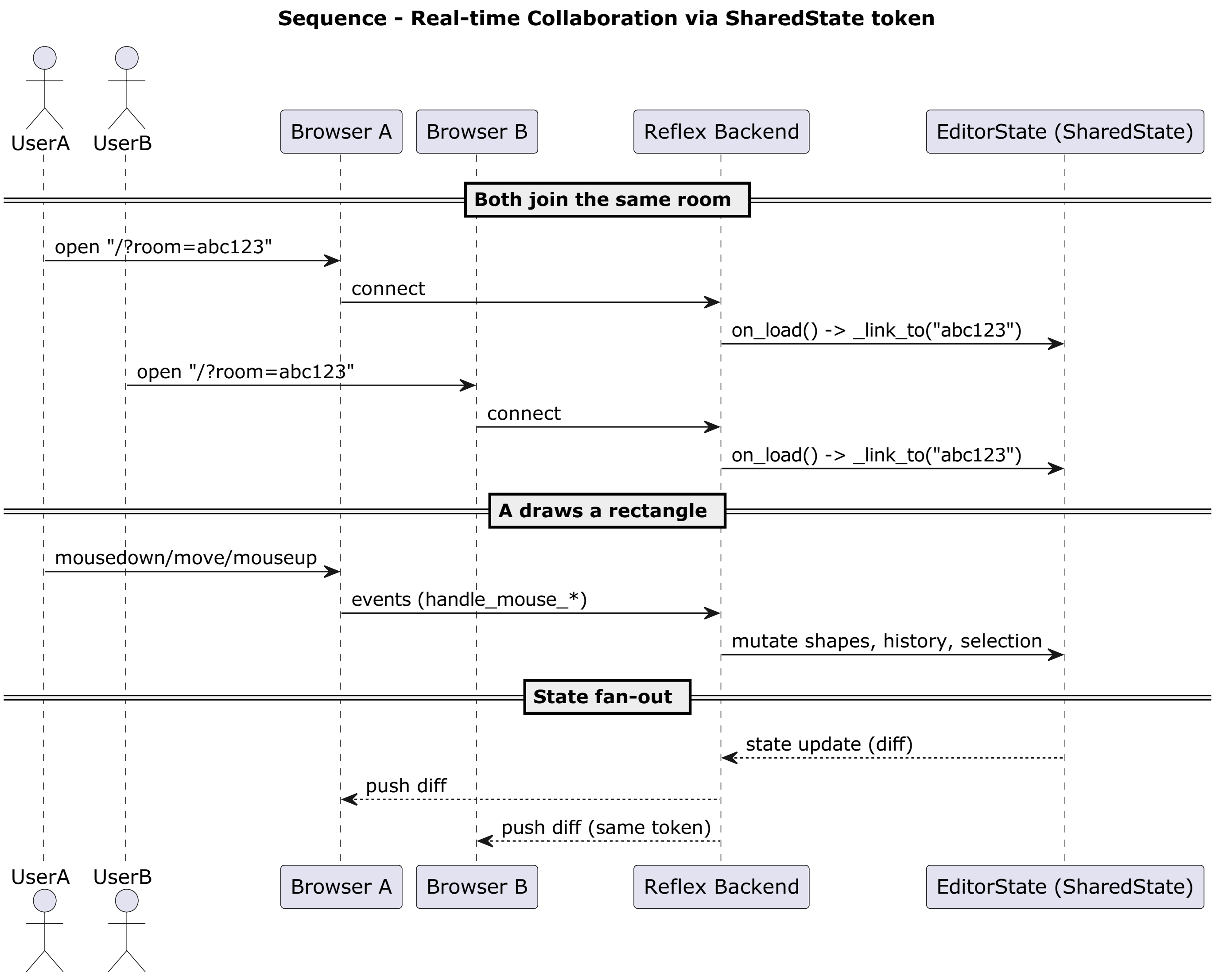

Sequence - Create Room -> Redirect -> Join SharedState
User
Topbar UI
Reflex Backend
EditorState (rx.SharedState)
Clipboard
Create a new room
click "Create Room"
create_room()
new_id = random(a-z0-9, 6)
full_url = origin + "/?room=" + new_id
set_clipboard(full_url)
toast("Link copied...") redirect("/?room=new_id")
Page reload / navigation
browser loads "/?room=new_id"
on_load()
room_param = query["room"]
room_id = room_param
safe_token = room_param.replace("_","-")
await _link_to(safe_token) (SharedState join)
state sync channel established

—

A2. Multi-user Collaboration: Same Room Shares the Same EditorState

The core idea is: if different users enter via the same `/?room=xxx`, `on_load()` links them to the same token, so they see the same shared `EditorState.shapes`, `selected_shape_id`, `pan_x/pan_y`, etc.

Sequence: UserA draws -> UserB syncs instantly



—

A3. Mapping MCP ops to Rooms and Integrating with Collaboration

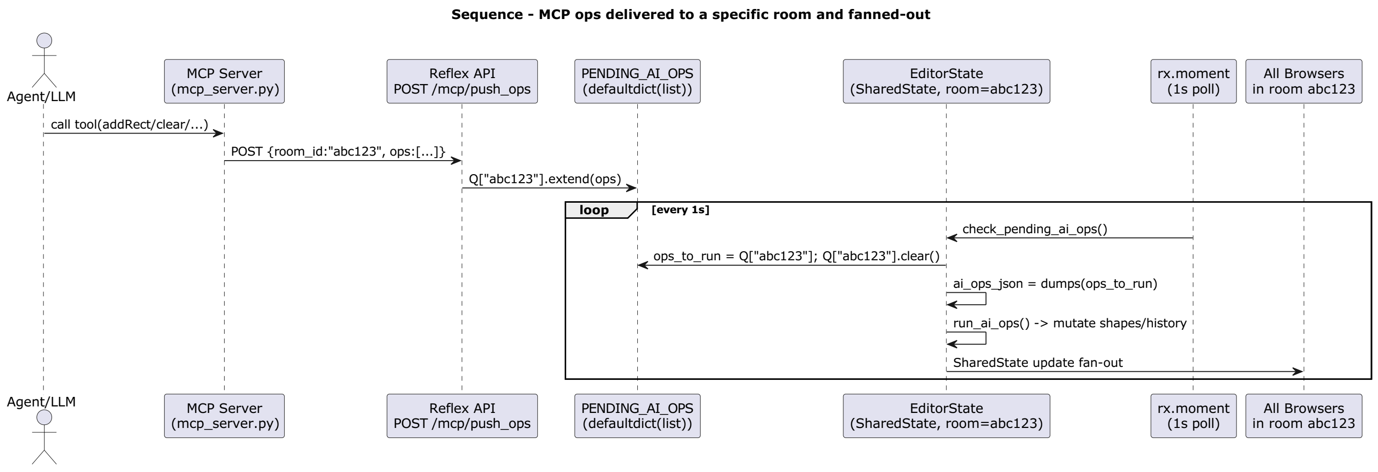
- `/mcp/push_ops` API (`codoc_in_vecdraw.py`):
- body: `{ room_id, ops }`
- `PENDING_AI_OPS[room_id].extend(ops)` (a global in-memory queue keyed by room)
- The frontend runs `EditorState.check_pending_ai_ops()` every `1000ms` (`rx.moment(interval=1000, ...)`)
- `check_pending_ai_ops()` uses:

- `target_room = self.room_id if self.room_id else "default"`

- It only pulls ops for the current room
- Immediately `clear()` after pulling to avoid duplicate processing
- Sets `self.ai_ops_json = json.dumps(ops_to_run)` and then calls `run_ai_ops()`

> Important limitation (as implemented): `PENDING_AI_OPS` is a process-local in-memory global dict. If you later switch to multi-worker / multi-instance, this queue will not automatically synchronize across processes; you will need a shared store such as Redis.

Sequence: MCP -> push_ops -> moment poll -> apply -> sync to all users in the room


Sequence - MCP ops delivered to a specific room and fanned-out
Agent/LLM
MCP Server (mcp_server.py)
Reflex API POST /mcp/push_ops
PENDING_AI_OPS (defaultdict(list))
EditorState (SharedState, room=abc123)
rx.moment (1s poll)
All Browsers in room abc123
call tool(addRect/clear/...)
POST {room_id:"abc123", ops:[...]}
Q["abc123"].extend(ops)
loop
[every 1s]
check_pending_ai_ops()
ops_to_run = Q["abc123"]; Q["abc123"].clear()
ai_ops_json = dumps(ops_to_run)
run_ai_ops() -> mutate shapes/history
SharedState update fan-out

—

B) Mouse Interaction Details: Select / Drag / Resize / Pan / Pencil / Text / Line

This section is organized strictly based on the actual branch behavior of `EditorState.handle_mouse_down/move/up()`.

B1. Coordinate System and Pan (Hand Tool)

- Canvas events first use `GET_COORDS_SCRIPT` in the frontend to get `raw_x/raw_y`.
- For tools other than the Hand tool, the backend computes:

- `x = raw_x - pan_x`

- `y = raw_y - pan_y` converting mouse positions into "world coordinates" (the coordinates stored in shapes).

Pan behavior (`current_tool == "hand"`)

- mouse down:

- `is_panning = True`
- `start_x = raw_x`
- `start_y = raw_y`
- return immediately (no selection/drawing)
- mouse move:
- `dx = raw_x - start_x`
- `dy = raw_y - start_y`
- `pan_x += dx; pan_y += dy`
- update `start_x/start_y = raw_x/raw_y`
- mouse up:
- if `is_panning`: set `is_panning = False` and return

—

B2. Select: Select the Top-most Shape Under the Cursor

When `current_tool == "select"`, mouse down scans shapes in reverse order (`reversed(self.shapes)`) so the top-most shape is hit-tested first.

Hit test rules:

- `rectangle/image/text/triangle/pencil`: bbox hit test
- `ellipse`: ellipse equation hit test `((x-cx)^2 / rx^2) + ((y-cy)^2 / ry^2) <= 1`
- `line`: rough hit test using line bbox + padding 5

On hit:

- `selected_shape_id = found_shape_id`
- `is_dragging = True`
- `drag_offset_x/y = x/y`
- `snapshot_shapes = deepcopy(shapes)` (for undo/redo)

On miss:

- `selected_shape_id = ""` (clear selection)

—

B3. Resize: Handle Hit Test First, Only for the Selected Shape

At the start of mouse down:

- If `selected_shape_id` exists:

- `handle = _get_handle_under_point(x,y,selected_shape)`
- If a handle is hit:
    - `active_handle = handle`
    - `is_dragging = True`
    - `snapshot_shapes = deepcopy(shapes)`
    - `drag_offset_x/y = x/y`
    - return (skip the select hit-test)

Handle types (based on `_get_handle_under_point()`):

- line: `start`, `end`
- other shapes: `n,s,e,w,ne,nw,se,sw` (hit radius ~6px)

—

B4. Drag/Resize Move Logic (Most Important Part)

On mouse move, if `is_dragging and selected_shape_id`:

- Compute `dx = x - drag_offset_x`, `dy = y - drag_offset_y`
- Update `drag_offset_x/y = x/y`
- Transform the selected shape:

(1) Resize: `active_handle != ""`

- line:
- handle `start`: move `x,y`
- handle `end`: move `end_x,end_y`
- non-line:
- handle contains `n`: `y += dy; height -= dy`
- handle contains `s`: `height += dy`
- handle contains `w`: `x += dx; width -= dx`
- handle contains `e`: `width += dx`
- If width/height becomes negative: auto-flip (convert width/height to positive, adjust x/y, and also flip the letters in `active_handle`)

> Current limitation (as implemented): resizing a `pencil` only changes its bbox (`x/y/width/height`), but pencil rendering uses `points/path_data`, so resizing a pencil stroke does not actually scale the stroke.

(2) Drag (Translate): `active_handle == ""`

- normal shapes: `x += dx; y += dy`
- line: also `end_x += dx; end_y += dy`
- pencil: add `dx/dy` to every point and recompute `path_data`

—

B5. Mouse Up: When History Is Recorded (undo/redo)

Mouse up falls into two cases:

1. Drawing mode (`is_drawing` and tool != select)

- pencil:

- if points > 1:

    - `past.append(snapshot_shapes)`
    - `future.clear()`
    - compute bbox (min/max)
    - generate `path_data = "M ... L ..."`
    - append shape
    - non-pencil:

- if `width > 2 or height > 2 or tool == line`:

    - push history
    - create shape (default fill/stroke)
    - for line: set start/end

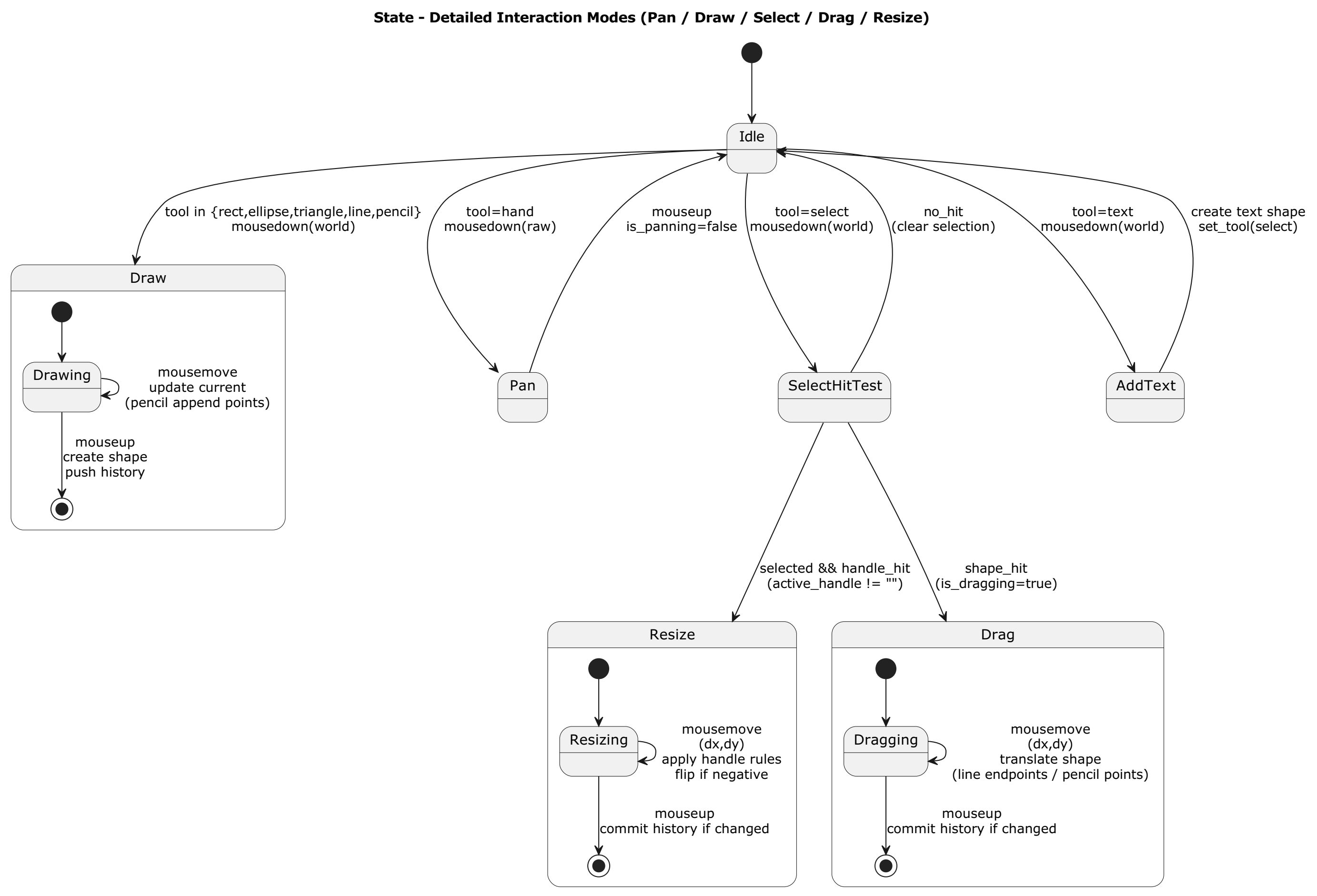
1. Drag/Resize mode (`is_dragging`)

- if `self.shapes != self.snapshot_shapes`:

- push history (`past += snapshot`)

—

B6. Interaction State Machine (More Detailed Version)



State - Detailed Interaction Modes (Pan / Draw / Select / Drag / Resize)
Idle
tool in {rect,ellipse,triangle,line,pencil} mousedown(world)
tool=hand mousedown(raw)
mouseup is_panning=false
tool=select mousedown(world)
no_hit (clear selection)
tool=text mousedown(world)
create text shape set_tool(select)
Draw
Drawing
mousemove update current (pencil append points)
mouseup create shape push history
Pan
SelectHitTest
AddText
selected && handle_hit (active_handle != "")
shape_hit (is_dragging=true)
Resize
Resizing
mousemove (dx,dy) apply handle rules flip if negative
mouseup commit history if changed
Drag
Dragging
mousemove (dx,dy) translate shape (line endpoints / pencil points)
mouseup commit history if changed

—

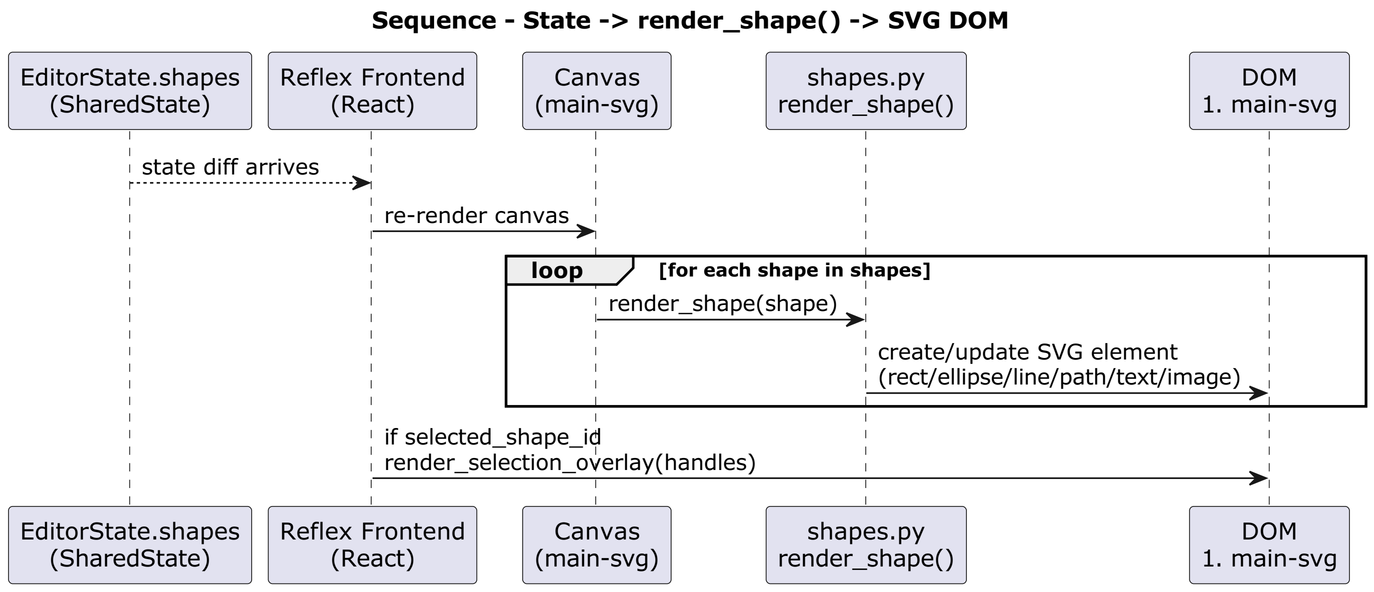
C) Render / Export Closed Loop: State -> SVG -> JS -> File Download

C1. Render Pipeline: EditorState.shapes -> shapes.py -> SVG DOM

- `components/canvas.py`:
- `rx.el.svg(id="main-svg")`
- `for shape in EditorState.shapes: render_shape(shape)`
- selection overlay: `render_selection_overlay(shape)` (in shapes.py)
- `render_shape()` (`components/shapes.py`):
- rectangle -> `<rect>`
- ellipse -> `<ellipse>`
- triangle -> `<polygon>`
- line -> `<line>`
- text -> `<text>`

- pencil -> `<path d=...>`
- image -> `<image href=upload_url ... preserveAspectRatio="none">`

Sequence: State update -> frontend re-render -> DOM ready


Sequence - State -> render_shape() -> SVG DOM
EditorState.shapes (SharedState)
Reflex Frontend (React)
Canvas (main-svg)
shapes.py render_shape()
DOM 1. main-svg
state diff arrives
re-render canvas
loop [for each shape in shapes]
render_shape(shape)
create/update SVG element (rect/ellipse/line/path/text/image)
if selected_shape_id render_selection_overlay(handles)

—

C2. Export Closed Loop: Topbar -> call_script -> export_canvas.js

Topbar Export menu:

- JSON:

- `EditorState.json_data_base64` (backend base64-encodes `json.dumps(shapes)`)

- frontend executes: `window.exportJSON(atob('...base64...'))`
- SVG:

- `window.exportSVG()`

- PNG:

- `window.exportPNG()`

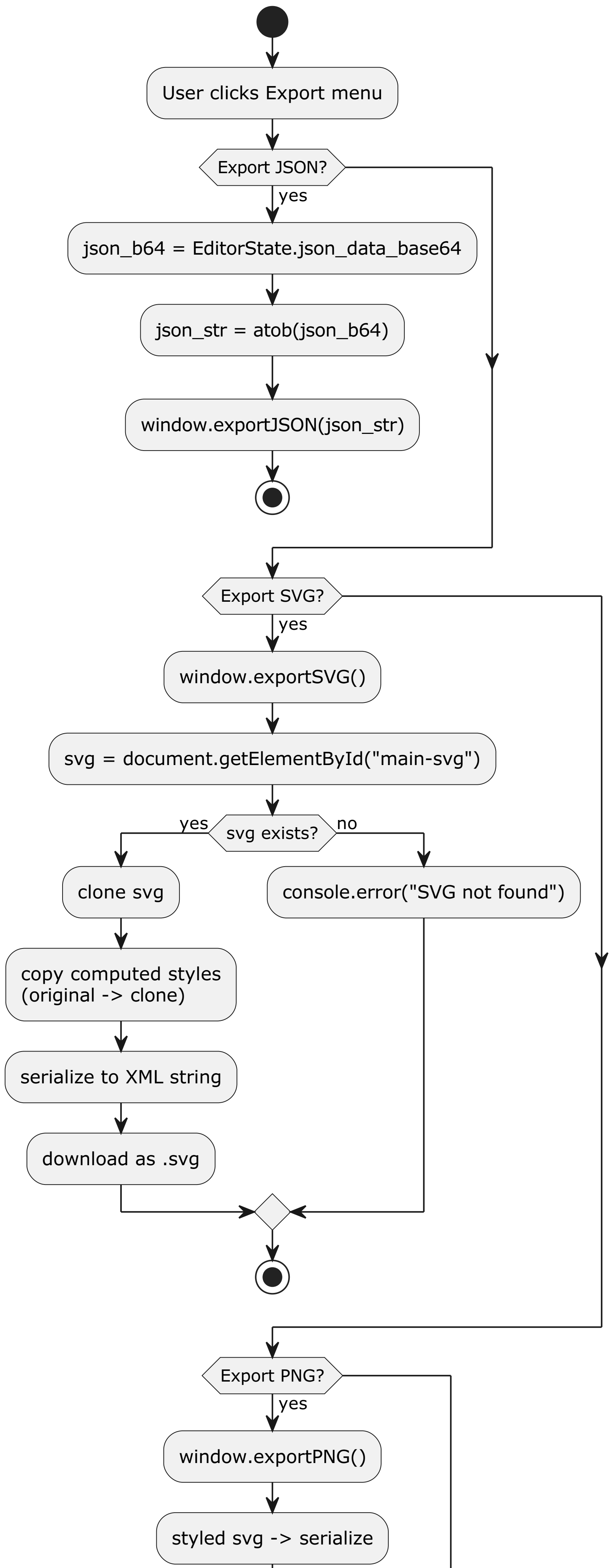
Core flow inside `assets/export_canvas.js` (SVG example):

1. `getStyledSVG("main-svg")`

- clone SVG
- traverse original vs clone and copy computed styles (so styles are not lost in export) 2. `serializeSVG(clonedSvg)` -> string 3. build data URL or blob 4. trigger download

Activity: frontend actions for Export SVG/PNG

Activity - export_canvas.js (SVG/PNG/JSON)

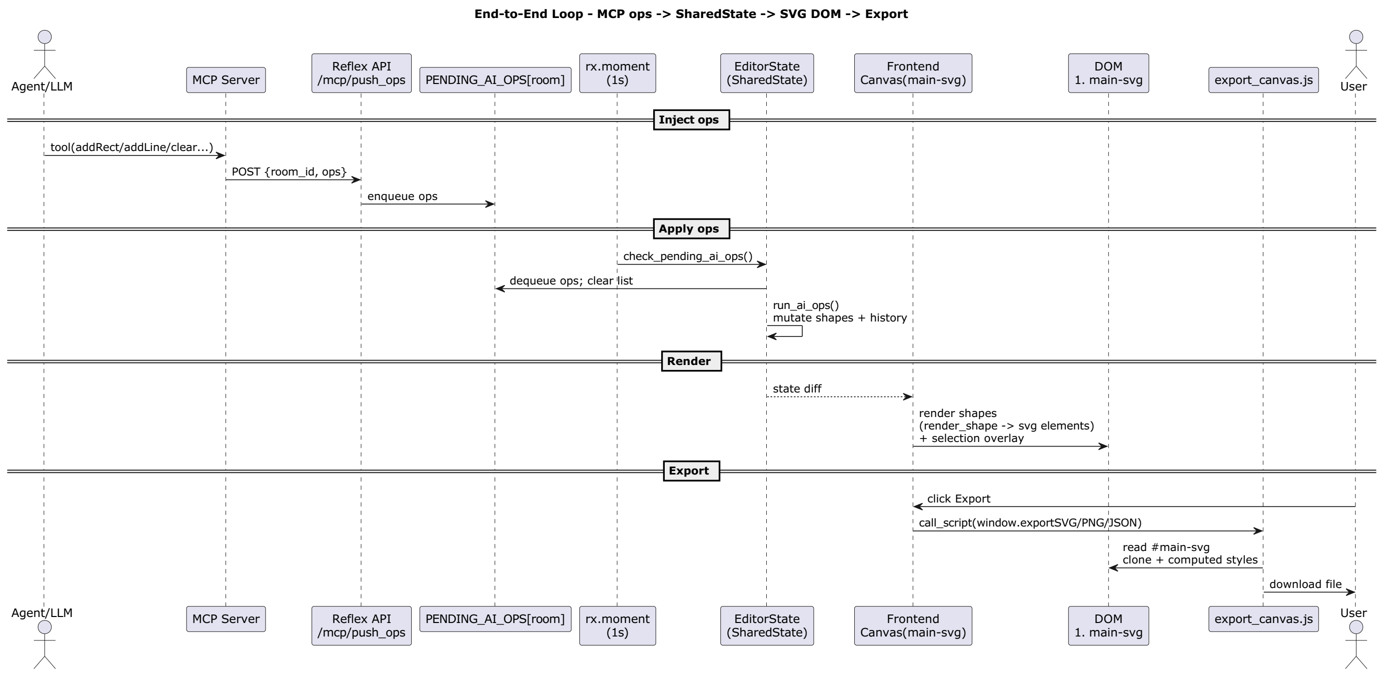


—

C3. Full Closed Loop Across Render/Export/AI (End-to-End)

This diagram ties together:

- MCP ops come in
- SharedState updates shapes
- SVG DOM re-renders
- Export reads the DOM and downloads


End-to-End Loop - MCP ops -> SharedState -> SVG DOM -> Export
Agent/LLM
MCP Server
Reflex API /mcp/push_ops
PENDING_AI_OPS[room]
rx.moment (1s)
EditorState (SharedState)
Frontend Canvas(main-svg)
DOM 1. main-svg
export_canvas.js
User
Inject ops
tool(addRect/addLine/clear...)
POST {room_id, ops}
enqueue ops
Apply ops
check_pending_ai_ops()
dequeue ops; clear list
run_ai_ops()
mutate shapes + history
Render
state diff
render shapes
(render_shape -> svg elements)
+ selection overlay
Export
click Export
call_script(window.exportSVG/PNG/JSON)
read #main-svg
clone + computed styles
download file