Nice — this is a great project and totally doable. Below I'll give a complete, practical design for the pipeline plus multiple ways to map React → **pini** (dictionary + more advanced ideas), implementation notes, a tiny working transform example you can drop into a Node script, verification & testing ideas, and a suggested roadmap to get it working quickly.

I'll assume **pini** exposes React-style components (e.g. `Pini.Button`, `Pini.Input`, or `import { Button } from 'pini'`) — the mapping approach below is flexible and configurable to whatever your real API is.

1) High-level architecture (end-to-end)

1. **Image input** — PNG/JPEG, high-res preferred.
2. **Preprocess** — normalize size, increase DPI if needed, denoise.
3. **Vision parsing** — detect UI elements, bounding boxes, segmentation, hierarchy (layout tree), OCR, color palette, fonts.
4. **UI AST creation** — generate an intermediate, device-agnostic UI AST (nodes like `Button`, `Text`, `Image`, `Input`, `Card`, `Row`, `Column`, `List`, with coords, style tokens).
5. **Semantic labeling** — LLM + rules convert visual detections to semantic labels (e.g. primary CTA, secondary link, nav item).
6. **React code generation** — generate clean JSX (component choices, props, children, styles or classNames, minimal inline styles).
7. **React → pini mapping** — transform JSX AST to pini AST/code using a mapping engine (dictionary + transforms).
8. **Preview & QA** — render in headless browser, screenshot, compare to original image (visual diff). Human review loop for edge cases.
9. **Store / telemetry** — store mappings, corrections, run analytics to improve mapping rules and ML models.

2) Vision parsing — what to extract

You want *as much* structured metadata from the image as possible. Key outputs:

- **Bounding boxes** for elements + segmentation mask (use it to crop or detect exact shape).
- **Element class**: button, label/text, input field, image, icon, card, list item, header, footer, navbar, avatar, checkbox, radio, toggle, switch, badge. (classifier confidence).
- **Text via OCR** (position, font-size estimate, weight, if possible).
- **Colors & palette** (dominant colors, background color). Map to your design tokens.
- **Spacing & layout**: margins, padding, alignment. Group elements into containers (horizontal/vertical stacks).
- **Repeating/recycler patterns**: repeated elements suggest a list or grid component.
- **Special states**: if it looks pressed, disabled (low contrast), etc — treat as metadata.

How to implement: segmentation (SAM), object detection (YOLO/GroundingDINO/Detectron2), OCR (Tesseract/Google Vision/Azure OCR), layout parsing (cluster bounding boxes — see next).

3) From boxes → layout tree (practical heuristics)

Goal: convert flat detections to a DOM-like tree.

Algorithm (outline):

1. Sort boxes by top coordinate.
2. For each box, find nearest container by spatial overlap or containment (if box A fully inside box B → child).
3. For boxes on same horizontal band (y overlap > threshold) cluster → `Row`. For vertical stacking form `Column`.
4. For repeating patterns (n similar boxes with similar size/spacing) create a `List` node with an `itemPrototype`.
5. Create nodes: `Container` (flex/grid), `Text`, `Button`, `Image`, `Input`, etc. Compute relative sizes (percent of parent).

These heuristics are robust and fast; refine with ML if needed.

4) React component selection

Given each AST node, select the best React component. Two complementary methods:

- **Rules/heuristics**: If node is rounded rectangle with centered text → `Button`. If underscored horizontal line + text box → `Input`. If repeated card shape → `Card` inside `List`.
- **Semantic matching**: Use component metadata (name, description, props, screenshot examples) and compute similarity between node description and component description (embedding search).

Output: a React AST / JSX string with imports, component tree, props, and minimal CSS/style tokens.

5) Mapping React → pini (the important part)

Overview

You want consistent UI across org apps by using `pini`. The general approach is:

- Build a **mapping engine** that transforms **React AST → pini AST** using a **mapping DB** (dictionary) + transformation functions.
- Prefer AST transforms (Babel or TS compiler API) rather than naive string replace. AST lets you reliably change component names, prop names, default props, and imports.

Dictionary idea (deep dive)

A dictionary file (JSON/YAML) describes component correspondences and prop transformations. It is the backbone and should be editable by devs/designers.

Schema (example):

```
{
  "Button": {
    "reactSelectors": ["Button", "button", "Btn"],
    "pini": {
      "import": "import { Button as PButton } from 'pini'",
      "componentName": "PButton"
    },
    "props": {
      "variant": {
        "mapTo": "kind",
        "transform": "value => value === 'primary' ? 'solid' : value"
      },
      "onClick": { "mapTo": "onPress" },
      "className": { "mapTo": "className" },
      "style": { "mapTo": "style" }
    },
    "styleTokenMapping": {
      "class:primary": "pini:primary",
      "class:secondary": "pini:secondary"
    },
    "priority": 10,
    "fallback": "Box"
  }
}
```

Fields explained

- `reactSelectors`: names to match in input JSX.
- `pini.import` & `pini.componentName`: how to import/rename.
- `props`: mapping per prop (mapTo, transform JS expression, default if missing).
- `styleTokenMapping`: map CSS classes or color tokens to pini theme tokens.
- `priority`: used when multiple components match.
- `fallback`: component to use if no precise mapping exists.

Benefits

- Transparent and editable by devs.
- Deterministic behavior & easy debugging.
- Versionable (commit mappings to repo).
- Can be extended to include example patterns, constraints, and screenshots of how it should look.

How to store it

- Start as a JSON file in repo (easy edit + PR flow).
- Then move to a small DB (Postgres) for multi-team editing & UI.
- Optionally add a vector DB for semantic search (for fuzzy mapping).

Runtime mapping algorithm (short)

1. Parse JSX into AST (Babel).
2. Walk AST; for each JSXElement, look up `name` in mapping dictionary (exact or alias).
3. If found:
    - Replace opening/closing names with pini component name.
    - Transform props according to `props` rules (rename, transform values, convert inline styles to tokens).
    - Add/adjust imports (avoid duplicates).
4. If not found:
    - Use fuzzy match via embeddings or fall back to primitive `Box`, `Text`, `Image`.
5. Emit pini code (format/Prettier).

6) Example transform code (Node.js using Babel)

This is a minimal but practical script that demonstrates parsing JSX and replacing `Button` with `PButton` per a dictionary. (In a real system you'd expand the prop mapping logic and import handling.)

```
// npm install @babel/parser @babel/traverse @babel/generator @babel/types
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generator = require("@babel/generator").default;
const t = require("@babel/types");

const mapping = {
  "Button": {
    "piniName": "PButton",
    "import": "import { Button as PButton } from 'pini';",
    "props": {
      "onClick": { "mapTo": "onPress" },
      "variant": { "mapTo": "kind", "transform": "v => (v === 'primary' ?
'solid' : v)" }
    }
  }
};

function transformJSX(source) {
  const ast = parser.parse(source, { sourceType: "module", plugins: ["jsx"]
});

  let addedImports = new Set();

  traverse(ast, {
    JSXElement(path) {
      const opening = path.node.openingElement;
      if (t.isJSXIdentifier(opening.name)) {
        const name = opening.name.name;
        if (mapping[name]) {
          const map = mapping[name];
          // Replace element name
          opening.name.name = map.piniName;
```

```
          if (path.node.closingElement)
            path.node.closingElement.name.name = map.piniName;

          // Transform props (simple rename)
          opening.attributes.forEach(attr => {
            if (t.isJSXAttribute(attr) && t.isJSXIdentifier(attr.name)) {
              const propName = attr.name.name;
              const propMap = map.props[propName];
              if (propMap) {
                attr.name.name = propMap.mapTo;
                // NOTE: transform functions could be applied here by
evaluating transform strings
              }
            }
          });

          addedImports.add(map.import);
        }
      }
    }
  });

  // Prepend imports if any
  const output = generator(ast, { quotes: "single" }).code;
  const imports = Array.from(addedImports).join("\n");
  return imports + "\n\n" + output;
}

// Example
const reactJSX = `
import React from 'react';
import { Button } from 'lib';

export default function Demo(){
  return <Button variant="primary" onClick={() => alert('hi')}>Sign
up</Button>
}
`;

console.log(transformJSX(reactJSX));
```

This prints transformed code with the `import` from `pini` and replaces `<Button>` → `<PButton>` and `onClick` → `onPress` (with additional work required to apply transform functions to values).

Production note: use `@babel/types` to create imports rather than string concat; this shortcut shows the approach.

7) More advanced/fuzzy mapping ideas (beyond a static dictionary)

- **Embeddings + semantic search**
  - Index component docs, examples, and visual screenshots. For each detected UI node, create a short text description and an embedding. Do semantic search

over component embeddings to get best-fit component candidates. Useful for ambiguous visuals.

- **Component ontology/graph**
  - Maintain a graph where nodes are components and edges capture "contains", "can-be-child", "has-props" relations. This helps with composite components (e.g., `NavBar` contains `NavItem`).
- **Intermediate DSL / UI-AST**
  - Convert image → UI-AST → target code (React / pini). The UI-AST is stable and can be target of multiple code generators.
- **Machine learned mapper**
  - Train a classifier (or few-shot LLM) that given an element feature vector (shape, text, colors) predicts the pini component and props. Use it when mapping evolves.
- **Template-based composition**
  - Keep example templates for common patterns (login page, pricing cards, list with image+text). Recognize pattern and instantiate parameterized templates in pini.
- **Human-in-the-loop (HITL)**
  - Present top-k mapping choices in the UI for quick developer approval. Capture corrections to automatically update dictionary or retrain models.

8) Mapping of CSS / style tokens

- **Color tokens**: extract palette and nearest token using CIEDE2000 distance. Map inline hex colors to theme tokens like `pini:primary`.
- **Spacing tokens**: round measured spacing to nearest token value (4px scale, 8px scale).
- **Typography**: extract font size/weight; map to your token system (e.g., `h1`, `body-md`). If exact font not available, fall back to brand font.
- **Icons**: use icon detection (SVG shape or small image); map to the nearest icon in pini's icon library by name/description.

9) Handling behaviors & interactions

- Images only show static state. For interactions (hover, onclick behavior, transitions):
  - Use semantic defaults for common components (e.g., Button → onClick mapped to `onPress` prop).
  - Allow the requestor to include behavior metadata in a companion JSON (if they can upload design handoff metadata like Figma metadata).
  - Human override for complex interactions.

10) Verification: pixel-matching and QA

- **Render** the generated pini code in a headless browser (Puppeteer).
- **Screenshot** it at same viewport & device pixel ratio.
- **Visual diff** with the original using tools like `pixelmatch`, `Resemble.js`, Backstop.js, or Percy. Report diffs and highlight mismatches.

- **Auto-fix steps**: if layout differs, adjust flex vs absolute rules, padding, fonts until within threshold. Then store that mapping as a higher-priority rule.

11) Dataset and continuous learning

- Build a dataset of `{image, targetReactCode, targetPiniCode}` pairs—start manually. Use it for:
  - Fine-tuning a code generation model or training a classifier for component mapping.
  - Few-shot examples for LLM prompts to generate accurate mappings.

12) Implementation stack & components

- **Backend**: Node.js (Babel for AST transforms), Python for CV (PyTorch models for segmentation/detection). Microservices or a single server with workers.
- **Vision models**: SAM (segmentation), Grounding DINO or Detectron2 (object detection), Tesseract/Google Vision (OCR).
- **LLM**: OpenAI / local LLM for text descriptions → generate JSX / mapping suggestions (use few-shot prompts with mapping dictionary examples).
- **DBs**: Git repo for mapping JSON; Postgres + UI for mapping editing; vector DB (Milvus/Pinecone) for embedding search.
- **Preview**: headless Chrome (Puppeteer) + local storybook server.
- **CI/test**: visual regression tests (Backstop/Percy).

13) Minimal MVP roadmap (weeks)

1. **Week 0–1**: Build extractor that detects rectangles, text (OCR), and returns bounding boxes + labels.
2. **Week 1–2**: Implement rule-based grouping → UI-AST. Create simple mapping dictionary for 10 core components (Button, Input, Text, Image, Card, NavBar, List, Icon, Badge, Modal).
3. **Week 2–3**: Implement AST-based transformer (Babel) and proof-of-concept mapping React→pini. Render preview and visual diff.
4. **Week 3–4**: Add LLM step to generate React JSX from UI-AST; add human-in-the-loop UI for quick corrections.
5. **Ongoing**: Add embedding-based fuzzy mapping, retraining, and analytics.

14) Example end-to-end (small, concrete)

Image: a simple login card with logo, Email input, Password input, and “Sign In” primary button.

Vision output (simplified):

```
[
  {"type":"Image","box":[10,10,120,80],"text":null},
```

```
  {"type":"Text","box":[30,100,300,140],"text":"Welcome back"},
  {"type":"Input","box":[30,150,300,190],"placeholder":"Email"},
  {"type":"Input","box":[30,200,300,240],"placeholder":"Password"},
  {"type":"Button","box":[30,250,300,290],"text":"Sign
In","variant":"primary"}
]
```

React JSX (generated):

```
import React from 'react';
import { Button, Input, Card } from 'ui-lib';

export default function LoginCard(){
  return (
    <Card style={{width: 360}}>
      <img src="logo.png" alt="logo" />
      <h2>Welcome back</h2>
      <Input placeholder="Email" />
      <Input placeholder="Password" type="password" />
      <Button variant="primary">Sign In</Button>
    </Card>
  );
}
```

pini mapping dictionary (snippet):

```
{
  "Button": {"piniName":"Pini.Button","import":"import { Button as
PiniButton } from 'pini';","props": {"variant": {"mapTo":"style"}}},
  "Input": {"piniName":"Pini.TextField","import":"import { TextField as
PiniTextField } from 'pini';","props":
{"placeholder":{"mapTo":"placeholder"}}},
  "Card": {"piniName":"Pini.Card","import":"import { Card as PiniCard }
from 'pini';","props": {}}
}
```

Transformed pini code (expected):

```
import React from 'react';
import { Card as PiniCard } from 'pini';
import { TextField as PiniTextField } from 'pini';
import { Button as PiniButton } from 'pini';

export default function LoginCard(){
  return (
    <PiniCard style={{ width: 360 }}>
      <img src="logo.png" alt="logo" />
      <h2>Welcome back</h2>
      <PiniTextField placeholder="Email" />
      <PiniTextField placeholder="Password" type="password" />
      <PiniButton style="primary">Sign In</PiniButton>
    </PiniCard>
  );
}
```

15) Edge cases & caveats

- **Exact pixel-match** is hard — fonts, antialiasing, and platform differences will cause differences. Aim for semantic parity and close visual parity, not 100% pixel perfection.
- **Animations/interactions** are not inferable from a static image; require metadata or human input.
- **Custom or composite components** might need developer-written adapters in mapping DB.
- **Ambiguity**: UI elements that look like a button might be a card — provide top-k choices and make the human select.

16) Monitoring + continuous improvement

- Log mapping decisions and human corrections.
- Use corrections to update the dictionary or re-train mapping model.
- Track error classes (wrong component, wrong prop mapping, wrong styling) and prioritize fixes.

TL;DR / Quick checklist you can start with now

- Build a **vision pipeline** that outputs bounding boxes, labels, OCR, palette.
- Create a **UI-AST** translator (grouping & layout heuristics).
- Produce **React JSX** from UI-AST (template + LLM if needed).
- Build a **mapping dictionary JSON** for the core components (Button, Input, Card, Nav). Put it under version control.
- Implement **AST-based transformer** (Babel) to convert React JSX → pini using the dictionary.
- Render, screenshot and run **visual diffs**; add HITL corrections.
- Add embeddings & telemetry for fuzzy matching & continuous learning.

If you want, I can:

- generate a starter **mapping JSON** for 15 common components tailored to the names you use in your real `pini` API (tell me a couple of canonical `pini` component names) — I'll produce a ready-to-use file and a Babel transform script that applies it; **or**
- produce a more polished **Babel transform** that handles import normalization + prop transforms + style token mapping.

Tell me which of those you want and I'll drop the exact code and mapping file ready to plug into your repo.