Got it — let's break this down. The diagram you shared looks like a **system architecture proposal** for how an **AI agent** can safely interact with your organization's services (here called *Pioneer service*) to process and analyze visual elements.

Components in the diagram

1. **Object storage**
   - Stores raw assets (like images, UI screenshots, or design files).
   - These are used as input for vision-based detection.
2. **Metadata storage**
   - Stores structured attributes about each element (e.g., type, size, color, relationships).
   - Probably acts like a searchable catalog or schema.
3. **Pioneer service**
   - Central service managing "PioneerElements".
   - You define a class here:
   - `class PioneerElement(BaseModel):`
   - `    id: UUID`
   - `    name: str`
   - `    lines: List[str/any/eta]  # attributes about the element`
   - `    code: str`
   - This service acts as a registry and mediator between storage and external agents.
4. **MCP (Mediator / Control Plane)**
   - Think of it as a **safety + governance layer** between AI and Pioneer.
   - It exposes **tools/APIs** (search, extract, list, etc.) but puts **safeguards** in place so AI cannot directly delete/edit sensitive data.
   - Tools listed:
     - `search_pioneer_db_code`
     - `search_pioneer_db_images`
     - `extract_element`
     - …and so on.
5. **AI Agent**
   - The system's intelligence layer.
   - It doesn't directly access your storage or database.
   - Instead, it can only call MCP's **safe tools** to fetch/search/extract data.
   - It is given a **system prompt** that explains its role ("an expert front-end developer") and restrictions (can't delete/modify, only analyze and propose).

Flow of operations

1. **Image/UI element lands in Object storage**.
2. **Pioneer service** processes it → extracts attributes (size, lines, code snippets, etc.).
3. Metadata about that element is stored in **Metadata storage**.
4. When an **AI Agent** needs to generate UI or analyze elements, it communicates with **MCP**, not directly with Pioneer.

5. **MCP** provides *safe tools* for the AI agent (like search, lookup, extract).
6. AI Agent calls these tools → retrieves relevant Pioneer elements → reasons on them → generates the requested output (e.g., new UI code).
7. **Safeguards** in MCP ensure the AI agent can't accidentally wipe or corrupt data (read-only + controlled API).

Why this idea is clever

- **Safety first**: LLMs are powerful but can hallucinate or make bad calls (like dropping tables). By putting MCP in the middle, you control what the AI can do.
- **Abstraction layer**: Pioneer service and MCP abstract away internal storage complexity, exposing only safe "developer-facing" tools.
- **Composable system**: Different AI agents (code generator, reviewer, designer) could all plug into MCP without touching core storage.
- **Future extensibility**: You can add more tools (like `search_pioneer_by_style`, `compare_elements`, `suggest_alternatives`) without retraining the agent — just update its prompt with new tool descriptions.

How this relates to your pini AI idea

This architecture is almost the **backend framework** you'd need:

- **Object storage** = screenshots of desired UI.
- **Pioneer service** = structured representation of UI components (buttons, inputs, navbars, etc.).
- **Metadata storage** = attributes (e.g., `component=Button, variant=Primary, text="Login"`).
- **MCP** = ensures the AI agent can only *query & generate mappings*, not mess with source-of-truth code.
- **AI Agent** = takes a screenshot, finds the right components, and outputs `pini` code.

✅ **In short**:
This diagram is showing a **guard-railed AI integration pattern**:

- AI is just a *consumer of safe APIs*, never a direct modifier of your system.
- You define a **domain-specific ontology (PioneerElement)** that lets AI work with structured components rather than raw pixels/code.
- MCP ensures the interaction is predictable, safe, and auditable.

Do you want me to **expand this into a concrete architecture for pini** (i.e., replace "Pioneer" with "pini" and show how screenshots → metadata → safe AI code generation → output code) so you can pitch/implement it for your org?

Nice — I'll translate the diagram into a full, concrete, implementable architecture for **pini** (replacing "Pioneer" with `pini`) so you can pitch it, hand it to engineers, or start building. I'll include data models, APIs, the Mediator/Control Plane (MCP) design, suggested tool interfaces for the AI agent, mapping strategies (dictionary + fuzzy), example payloads, sample transform code, safety/audit ideas, and a rollout roadmap.

1 — High-level objective (one-liner)

Given an image/screenshot, produce verified **pini** UI code that reproduces the UI while keeping the system safe, auditable, and maintainable.

2 — Principal components

1. **Object Storage**: raw input assets (screenshots, designer exports) and generated assets (rendered screenshots, thumbnails).
2. **Metadata DB**: stores parsed element metadata (type, bounding box, style tokens, confidence, provenance).
3. **pini Service (registry)**: authoritative store for `PiniElement` objects and component metadata, plus search APIs.
4. **MCP (Mediator / Control Plane)**: single entry point for AI agents — exposes safe, granular tools (search, extract, preview) and enforces read-only/safe mutations.
5. **AI Agent(s)**: runs the vision + LLM pipeline; calls MCP tools to retrieve data and get/generate code; returns code to user or a staging repo.
6. **Render & Visual Diff Service**: renders generated code headlessly, compares to original image and scores similarity.
7. **Human-in-the-loop UI**: preview, corrections UI; accepts edits that can update mapping DB or create new mapping rules.
8. **Mapping DB**: dictionary for mapping React components → pini components, plus transformation functions.
9. **Audit & Telemetry**: logs all agent actions and tool calls; stores corrections and metrics.

3 — Core data models

`PiniElement` (schema)

```
type UUID = string;

interface BoundingBox { x: number; y: number; w: number; h: number; }

/** authoritative record for an element discovered in an image or created
by developers */
interface PiniElement {
  id: UUID;
```

```
  sourceImageId: UUID;
  name?: string;            // semantic label e.g. "Primary CTA"
  type: string;             // e.g. "Button", "Text", "Input", "Card",
"Icon"
  bbox: BoundingBox;
  text?: string;            // OCR text if any
  styleTokens?: {           // mapped tokens, optional
    color?: string;         // theme token (e.g., "brand.primary")
    typography?: string;    // token e.g., "h2-sm"
    spacing?: string;
    radius?: string;
  };
  confidence: number;       // detection confidence
  code?: string;            // canonical React/pini snippet or prototype
  createdAt: string;
  updatedAt?: string;
  provenance?: {
    detectedBy: string;     // model/version
    humanVerified: boolean;
    verifierId?: string;
  };
}
```

4 — MCP: role, design, and tools

MCP responsibilities

- Provide a minimal, strictly controlled toolset to AI agents (no raw DB queries).
- Enforce RBAC, rate limits, and audit logging.
- Validate and sanitize requests/responses.
- Offer composite, safe operations (search, extract, preview render) rather than exposing destructive ops.

Example MCP toolset (REST/JSON-RPC)

1. search_elements

- Input: `{ query: string, filters?: {type?:string, colorToken?:string, textContains?:string}, limit?: number }`
- Output: `[{id, type, bbox, text, styleTokens, snippetPreviewUrl, confidence}]`
- Use-case: agent finds exemplar components.

2. fetch_element

- Input: `{ id: UUID }`
- Output: full `PiniElement` record (read-only).

3. extract_element_from_image

- Input: `{ imageId: UUID, bbox?: BoundingBox }`
- Output: `PiniElement` objects detected (with confidence). This operation runs vision models.

4. render_code_preview

- Input: `{ code: string, viewport: {w,h,scale?} }`
- Output: `{ renderId, screenshotUrl, score?: number }` (score computed by visual diff service optionally)

5. compare_rendered_to_source

- Input: `{ sourceImageId, renderedImageId }`
- Output: `{ diffPercent, heatmapUrl, mismatches: [{region, reason}...] }`

6. propose_mapping

- Input: `{ reactComponentDescriptor: string }`
- Output: candidate mapping(s) from mapping DB with confidence.

7. request_human_approval

- Input: `{ workflowId, changes, reviewerGroup }`
- Output: approval ticket id — opens human-in-loop.

MCP returns only URLs to rendered assets; actual raw data access is gated.

Tool-call policy (for the AI agent)

- Each call is recorded with `agentId`, `promptContext`, and timestamp.
- No write-capability in default policy; writes must pass an approval step.
- Agents can only act within a `scoped-api-key` scope (e.g., read-only for prod).

5 — Vision + AST pipeline (high level)

1. **Preprocess image** (normalize DPI, crop, upscale if needed).
2. **Run detectors**: object detection (buttons, inputs, cards), segmentation for shapes, OCR for text.
3. **Cluster into containers** and produce UI-AST (tree of nodes). Each node includes bbox, type, text, style sample.
4. **Produce a semantic description** per node (short human-style sentence).
5. **LLM stage**: optionally convert the UI-AST or node descriptions to React JSX code (component selection, props, style tokens). Use mapping DB for component name hints.
6. **React → pini transform**: apply the mapping engine (dictionary or fuzzy mapper) to create pini code.
7. **Render & diff** and iterate.

6 — Mapping strategies (detailed)

6.1 Dictionary (canonical)

A JSON/YAML mapping file that maps `ReactComponent` → `pini` with prop and style transformations. This is the recommended first step.

Example mapping JSON

```
{
  "Button": {
    "reactSelectors": ["Button","button","Btn"],
    "pini": {
      "import": "import { Button as PButton } from 'pini';",
      "componentName": "PButton"
    },
    "props": {
      "onClick": { "mapTo": "onPress" },
      "variant": { "mapTo": "kind", "transform": "v => v === 'primary' ?
'solid' : v" },
      "disabled": { "mapTo": "disabled" }
    },
    "styleTokenHints": {
      "backgroundColor": { "nearestToken": true },
      "borderRadius": { "nearestToken": true }
    },
    "priority": 20,
    "fallback": "Box"
  }
}
```

Dictionary pros

- Transparent, versionable via Git.
- Easy to review & override.
- Deterministic mapping.

Dictionary cons

- Needs maintenance as `pini` evolves and new React patterns appear.

6.2 Fuzzy mapping (embedding + search)

- Index component docs, examples, and screenshots as embeddings in a vector DB.
- For an unknown or ambiguous React component descriptor, compute an embedding and run a semantic search to get best candidates (top-K).
- Use a scorer combining semantic similarity and usage frequency.

When to use: ambiguous cases, composite components, or when the dictionary lacks an exact match.

6.3 Hybrid approach (recommended)

- Primary: dictionary lookup (fast + deterministic).
- Secondary: if no confident match, run embedding-based fuzzy match and present top-3 to the human-in-loop or agent for selection.

- Tertiary: fallback to primitives (`Box`, `Text`, `Image`) plus a `suggestMapping` ticket for maintainers.

7 — React-to-pini transform engine (implementation notes)

- Work on ASTs (Babel for JS/TSX) — never do string replace.
- Steps:
    1. Parse source to AST.
    2. Walk nodes to identify components to map.
    3. Consult mapping DB (dictionary). If not found, call MCP `propose_mapping` to get candidates.
    4. Replace JSX identifiers, transform props via mapping rules (allow JS transforms), map inline styles to theme tokens (CIEDE2000 nearest color approach), and adjust imports.
    5. Emit formatted code; run linter/formatters.

Example transform snippet (pseudocode) — simplified:

```
const ast = parseJSX(src);
for (node of ast.jsxElements) {
  const name = getName(node);
  const map = mappingDB.find(name) || propose_mapping(name);
  if (map) {
    renameElement(node, map.pini.componentName);
    transformProps(node, map.props);
    addImport(map.pini.import);
  } else {
    wrapWithBox(node);
  }
}
const out = generate(ast);
```

8 — Example end-to-end request & payloads

User submits:

`POST /api/v1/generate` body:

```
{
  "imageId": "img-123",
  "targetFramework": "pini",
  "options": {
    "viewport": { "w": 390, "h": 844 },
    "pixelDiffThreshold": 3.0,
    "humanApproval": true
  },
  "metadata": {
    "projectId": "proj-abc",
```

```
    "requestedBy": "kanika@example.com"
  }
}
```

MCP workflow (simplified)

1. MCP calls `extract_element_from_image(imageId)` → returns UI-AST (list of `PiniElement`s).
2. For each `PiniElement`, agent calls `search_elements` to find matching `pini` components and examples.
3. Agent calls LLM to generate React JSX (or directly generate pini code if you prefer) — LLM prompt includes element descriptions & top mapping suggestions.
4. Transform React → pini (AST transform).
5. Call `render_code_preview` with the generated code.
6. Get `screenshotUrl`; call `compare_rendered_to_source(sourceImageId, renderedImageId)` → get `diffPercent`.
7. If `diffPercent` < threshold, produce draft PR with code; else request human approval & present UI diff.

9 — Sample LLM prompt templates

A — Generate React code (short)

```
SYSTEM: You are an expert frontend engineer. Produce clean React JSX for
the following UI AST. Use semantic components only; prefer library
components when available. Only output code, no explanation.

USER: UI AST:
1) {type: "Image", bbox:..., src: "..."}
2) {type: "Text", text: "Welcome back", typography: "h2-sm"}
3) {type: "Input", placeholder: "Email"}
4) {type: "Input", placeholder: "Password", type: "password"}
5) {type: "Button", text: "Sign in", variant: "primary"}

Produce React code (no imports necessary if not required).
```

B — Map to pini (post-React)

```
SYSTEM: You are a mapping assistant. Convert JSX to pini using the
following mapping rules (give exact JSX output and imports). Mapping rules:
Button->Pini.Button (onClick->onPress, variant->kind), Input-
>Pini.TextField...
USER: [paste React JSX]
```

10 — Safety, governance, and audit

- **Read-only-by-default**: MCP does not expose destructive endpoints to AI. Any write must be approved.
- **Scoped agent keys**: limit the scope of each agent. Production agents use stricter scopes and require manual enablement.

- **Audit logs**: every tool call traced (agentId, prompt text, returned results). Store logs in append-only storage.
- **Human approval gating**: any code that would be merged into shared libs or design tokens requires a review.
- **Explainability**: include a mapping report with each generated artifact explaining mapping decisions (which dictionary rule was used, fuzzy-match score, color token nearest match). This helps reviewers understand decisions quickly.

11 — Render & visual diff (practical)

- Render generated code inside an isolated preview environment (Storybook or small container).
- Use headless Chrome/Puppeteer to screenshot with the same viewport/dpr.
- Use pixelmatch or `Resemble.js` to compute percent-diff and generate heatmaps.
- If layout mismatch > threshold, surface the diff to a reviewer with suggested fixes (e.g., update margin, font size token).

12 — Human-in-the-loop UX (what to show)

- Original image (left), generated render (center), heatmap/diff (right).
- For each mapped element, show:
  - source bbox and detected type/confidence,
  - chosen `pini` component and mapping rule used (click to edit),
  - token mapping suggestions for color/spacing/typography (click to accept),
  - code snippet preview inline and option to patch code and re-render.
- Approve button creates a PR or copies code to target repo.

13 — Devops / infra considerations

- Vision models (SAM, detector) run in a GPU-enabled service (K8s or cloud ML infra).
- LLM calls may use hosted or private LLMs; cache prompts & top-K completions.
- Mapping DB stored in versioned Git + metadata in Postgres for quick queries.
- Use message queue (Kafka/RabbitMQ) to orchestrate long-running tasks (render, visual diff).
- Secure object storage (S3) with signed URLs for MCP to provide to agents/humans.

14 — Rollout roadmap (MVP → production)

Phase 0 (week 0): basic extraction only

- Implement `extract_element_from_image` and `fetch_element`.

- Store `PiniElement` entries in DB (manually created dataset).

Phase 1 (weeks 1–3): rule-based pipeline + dictionary mapping

- Build mapping DB + AST transform (Babel).
- Add `render_code_preview` + simple visual diff.
- Deliver simple Human-in-Loop UI for approvals.

Phase 2 (weeks 4–8): LLM integration + fuzzy mapping

- Add LLM steps to propose React code from AST and map candidates via embedding search.
- Implement top-k suggestions and automated candidate ranking.

Phase 3 (ongoing): scale & automation

- Expand mapping DB, add auto-updates from human corrections.
- Improve detection models, train component classifier, add CI checks and auditing.

15 — Example artifacts you can start with today

1. **Starter mapping JSON** (I can generate a 15-component JSON for your `pini` naming schema).
2. **Small Babel transform** that replaces `Button` → `Pini.Button` and maps props — drop-in to build pipeline.
3. **LLM prompt templates** for React and mapping steps.
4. **MCP API spec** (OpenAPI) for devs to implement.

Quick sample — minimal mapping JSON + transform snippet

mapping.json

```
{
  "Button": {
    "reactSelectors": ["Button","button"],
    "pini": {"componentName":"Pini.Button","import":"import { Button as
PiniButton } from 'pini';"},
    "props":
{"onClick":{"mapTo":"onPress"},"variant":{"mapTo":"kind","transform":"v =>
v === 'primary' ? 'solid' : v"}}
  },
  "Input": {
    "reactSelectors": ["Input","TextField","input"],
    "pini": {"componentName":"Pini.TextField","import":"import { TextField
as PiniTextField } from 'pini';"},
```

```
    "props":
{"placeholder":{"mapTo":"placeholder"},"type":{"mapTo":"type"}}
  },
  "Card": {
    "reactSelectors": ["Card"],
    "pini": {"componentName":"Pini.Card","import":"import { Card as
PiniCard } from 'pini';"}
  }
}
```

Simplified Babel-like transform (pseudocode/JS)

```
// parse -> walk -> replace JSXIdentifiers according to mapping.json ->
update props
// use @babel/parser, traverse, generator in real code

function applyMapping(ast, mapping) {
  traverse(ast, {
    JSXElement(path) {
      const name = path.node.openingElement.name.name;
      const entry = findMappingEntry(name, mapping);
      if (!entry) return;
      const newName = entry.pini.componentName;
      path.node.openingElement.name.name = newName;
      if (path.node.closingElement) path.node.closingElement.name.name =
newName;
      // props
      path.node.openingElement.attributes.forEach(attr => {
        if (!attr.name) return;
        const pm = entry.props && entry.props[attr.name.name];
        if (pm) attr.name.name = pm.mapTo;
      });
      // imports handled separately
    }
  });
}
```

Final suggestions & next steps (pick one)

1. I can generate a starter `mapping.json` for 15 most common components in your `pini` API and a ready-to-run Babel transform script (Node.js) to test converting React → pini locally.
2. I can produce an `MCP` OpenAPI (YAML) spec that your backend engineers can implement and connect to vision and render services.
3. I can craft the LLM prompt suite (templates + few-shot examples) optimized to generate React code from UI-AST and then mapping prompts to produce `pini` code.

Tell me which artifact you want first (I'll *immediately* produce it here) — mapping JSON, Babel transform script, or MCP OpenAPI spec — and I'll drop it into the next message.

Perfect — let's lock in the **LLM prompt suite** for pini.
I'll give you three reusable templates you can plug into your orchestration layer (e.g. MCP → LLM call). Each is optimized for a different stage of the pipeline:

📄 pini LLM Prompt Suite

1. UI-AST → React JSX (first code generation stage)

Turns the detected UI structure (AST from vision + OCR) into **clean React JSX**.

```
SYSTEM:
You are a senior frontend engineer.
Convert the given UI AST into clean, idiomatic React JSX.
- Use semantic components only (Button, Input, Text, Card, Image, etc).
- Do NOT invent styles; only use props explicitly provided.
- Never include explanations — output JSX code only.

USER:
UI AST:
[
  { "type":"Image","src":"hero.png","bbox":{...}},
  { "type":"Text","text":"Welcome back","typography":"h2-sm"},
  { "type":"Input","placeholder":"Email"},
  { "type":"Input","placeholder":"Password","inputType":"password"},
  { "type":"Button","text":"Sign in","variant":"primary"}
]
```

👉 **Expected Output:**

```
<>
  <Image src="hero.png" alt="Hero" />
  <Text variant="h2-sm">Welcome back</Text>
  <Input placeholder="Email" />
  <Input type="password" placeholder="Password" />
  <Button variant="primary">Sign in</Button>
</>
```

---

2. React JSX → pini JSX (mapping stage)

Takes React code and applies mapping rules (dictionary + fuzzy hints).
Keeps output strictly in `pini` syntax.

```
SYSTEM:
You are a mapping assistant.
Convert JSX to use pini components according to mapping rules.
- Replace React components with their pini equivalents.
- Apply prop transformations (e.g. onClick -> onPress).
- Ensure correct pini imports.
- Output only valid JSX + imports.

USER:
React JSX:
<>
  <Button variant="primary" onClick={handleSubmit}>Sign in</Button>
  <Input placeholder="Email" />
```

```
</>

Mapping rules (JSON):
{
  "Button": {
    "pini": {"componentName":"Pini.Button","import":"import { Button as
PiniButton } from 'pini';"},
    "props": {"onClick":{"mapTo":"onPress"},"variant":{"mapTo":"kind"}}
  },
  "Input": {
    "pini": {"componentName":"Pini.TextField","import":"import { TextField
as PiniTextField } from 'pini';"},
    "props":
{"placeholder":{"mapTo":"placeholder"},"type":{"mapTo":"type"}}
  }
}
```

👉 Expected Output:

```
import { Button as PiniButton, TextField as PiniTextField } from 'pini';

<>
  <PiniButton kind="primary" onPress={handleSubmit}>Sign in</PiniButton>
  <PiniTextField placeholder="Email" />
</>
```

---

3. Code Verification + Self-Correction (optional loop)

Asks the LLM to **check and repair** generated pini code before render.

```
SYSTEM:
You are a strict code reviewer for pini.
Check the provided JSX for errors against pini API.
- Ensure correct component names and imports.
- Ensure only valid props are used (remove unsupported props).
- Fix obvious mistakes.
- Output only corrected JSX + imports.

USER:
Code:
import { Button as PiniButton, TextField as PiniTextField } from 'pini';

<>
  <PiniButton variant="primary" onClick={submitForm}>Login</PiniButton>
  <PiniTextField label="Email" placeholder="Enter email" />
</>
```

👉 Expected Output:

```
import { Button as PiniButton, TextField as PiniTextField } from 'pini';

<>
  <PiniButton kind="primary" onPress={submitForm}>Login</PiniButton>
  <PiniTextField placeholder="Enter email" />
</>
```

🔗 How to use

- **Stage 1**: Call **Prompt 1** (UI-AST → React JSX).
- **Stage 2**: Feed output into **Prompt 2** with mapping JSON (fetched from MCP).
- **Stage 3** (optional): Run **Prompt 3** to enforce correctness before rendering & diff.

You can chain these inside MCP tool orchestration (each is just a different `system` + `user` template).

Want me to now **write the actual MCP orchestration flow** that plugs these prompts together (pseudo-code for calling LLM with these templates and tool calls)? That would show exactly how the AI agent cycles through extraction → generation → mapping → correction → render.

Great 👍 let's design the **MCP orchestration flow** for pini.
This is the control loop that ties together **vision → React → pini → verification**.

I'll show you the flow in structured pseudo-code that you could implement in Python/TypeScript (or any language calling MCP + LLMs).

🛠 MCP Orchestration Flow for pini

Step 0: Input

- User (designer/dev) uploads **UI screenshot**.
- Object Storage saves it.
- Pioneer/Pini Vision service extracts **UI AST** (structured JSON of elements).

Example AST:

```
[
  {"type":"Image","src":"hero.png"},
  {"type":"Text","text":"Welcome back","typography":"h2-sm"},
  {"type":"Input","placeholder":"Email"},
  {"type":"Input","placeholder":"Password","inputType":"password"},
  {"type":"Button","text":"Sign in","variant":"primary"}
]
```

Step 1: UI AST → React JSX

```
react_jsx = LLM.call(
  system=PromptSuite.react_generator_system,
  user={"UI AST": ast_json}
```

```
)
```

Step 2: Fetch Mapping Rules

```
mapping_rules = MCP.call("search_pioneer_db_code", query="component
mapping")
# returns JSON dictionary of React → pini component mappings
```

Step 3: React JSX → pini JSX

```
pini_jsx = LLM.call(
  system=PromptSuite.pini_mapper_system,
  user={
    "React JSX": react_jsx,
    "Mapping rules": mapping_rules
  }
)
```

Step 4: Code Verification & Self-Correction

```
verified_code = LLM.call(
  system=PromptSuite.pini_verifier_system,
  user={"Code": pini_jsx}
)
```

Step 5: Optional Diff (Safety Net)

```
# render React JSX (baseline) → screenshot
# render Pini JSX (candidate) → screenshot
# run image-diff tool → highlight mismatches

diff_report = MCP.call("compare_rendered_images", {
    "react_img": render(react_jsx),
    "pini_img": render(verified_code)
})

if diff_report["match"] < 95%:
    # loop back to Stage 3 with feedback
    pini_jsx = LLM.call(
        system=PromptSuite.pini_mapper_system,
        user={
          "React JSX": react_jsx,
          "Mapping rules": mapping_rules,
          "Feedback": diff_report
        }
    )
```

Step 6: Return Final Pini Code

```
return {
    "status": "success",
    "pini_code": verified_code,
    "preview": render(verified_code),
    "diff_report": diff_report
}
```

🔄 Flow Summary

1. **Extract UI AST** (vision layer).
2. **Generate React JSX** (Prompt 1).
3. **Fetch dictionary/mapping from MCP** (React → Pini).
4. **Transform JSX to pini** (Prompt 2).
5. **Verify + correct code** (Prompt 3).
6. **Image-diff validation** (optional loop for high fidelity).
7. **Deliver ready-to-use pini JSX code.**

🚀 Key Advantages

- **Safe by design**: LLM never touches raw DB, only queries MCP tools.
- **Composable**: you can swap the LLM, mapping rules, or diff-checker without rewriting flow.
- **Feedback loop**: image-diff ensures “pixel perfect” match, not just structural mapping.
- **Reusability**: same orchestrator can work for web, mobile, or future pini variants.

Would you like me to now **expand the mapping storage design** (how the dictionary of React→pini components is stored, queried, and extended) — so MCP can serve it dynamically instead of hardcoding? That’s the real backbone of scaling this system.