perplexity

ok now i want you create a guide on the screens for the practioner portal - i will provide some HTML mock ups - I want you to give clear guidence on the code to be used and no mixed versions etc - lay out what backend service will display on the screen and provide a full run down of it

I'll analyze the provided HTML mockups and create a structured guide for the practitioner portal screens, focusing on technical implementation and backend integration. Here's the comprehensive breakdown:

AI Practitioner Portal: Screen Architecture & Technical Guide

Core UI Principles

1. **Consistent Component Library**: All UI elements use a centralized design system with TypeScript React components
2. **State Management**: Redux Toolkit for cross-component state sharing
3. **Real-Time Collaboration**: WebSocket integration for multi-user editing
4. **Component Versioning**: Semantic versioning enforced through CI/CD pipelines

1. Dashboard & Workspace Selection

HTML Mockup Features: Project grid, quick filters, recent files, search bar

Frontend Implementation

```
// Dashboard.tsx
import { ProjectGrid, QuickFilterBar, SearchInput } from '@canvas-ui/core';

const Dashboard = () => (
  <div className="bg-canvas-dark p-6">
    <SearchInput onSearch={(query) => api.searchProjects(query)} />
    <QuickFilterBar
      filters={['Recent', 'LLM', 'Database', 'Templates']}
      defaultValue="Recent"
    />
```

```
    <ProjectGrid
      projects={useAppSelector(selectRecentProjects)}
      onProjectSelect={(id) => navigate(`/workspace/${id}`)}
    />
  </div>
);
```

Backend Services

- **GET /api/projects**: Returns paginated projects with metadata

```
{
  "id": "proj_abc123",
  "name": "Customer Support Bot",
  "lastModified": "2025-06-09T14:30:00Z",
  "components": ["llm-gpt4", "db-pgvector"],
  "template": "support-agent-v2"
}
```

Key Integration Points

- Project metadata synchronization via WebSocket
- Search uses ElasticSearch index of project contents
- Recent projects sorted by ML model predicting relevance

2. Component Library Browser

HTML Mockup Features: Categorized components, search, dependency visualization

Frontend Implementation

```
// ComponentLibrary.tsx
const ComponentLibrary = () => {
  const [category, setCategory] = useState('inputs');

  return (
    <div className="grid grid-cols-[240px_1fr]">
      <CategorySidebar
        categories={COMPONENT_CATEGORIES}
        onSelect={setCategory}
      />
      <ComponentGrid
        components={filteredComponents}
        onDragStart={(e, comp) => e.dataTransfer.setData('component', JSON.stringify(comp
      />
    </div>
  );
};
```

Backend Services

- **GET /api/components**: Returns available components

```
{
  "id": "llm-gpt4",
  "type": "llm",
  "inputs": ["prompt", "temperature"],
  "outputs": ["completion"],
  "compatibility": [">=v2.3.0"],
  "documentation": "/docs/components/llm/gpt4"
}
```

Validation Rules

- Component version checking against project requirements.txt
- Visual dependency resolution using D3-force simulation

3. Agent Builder Canvas

HTML Mockup Features: Drag-and-drop interface, connection lines, component properties

Frontend Implementation

```
// AgentCanvas.tsx
const AgentCanvas = () => {
  const { components } = useAppSelector(selectCurrentProject);

  return (
    <ReactFlow
      nodes={components}
      edges={connections}
      onConnect={(params) => api.createConnection(params)}
      nodeTypes={{
        llm: LLMNode,
        database: DatabaseNode,
        input: InputNode
      }}
    />
  );
};
```

Backend Services

- **POST /api/canvas/state**: Saves canvas layout state

```
{
  "nodes": [
    {
      "id": "node_1",
```

```
      "type": "llm",
      "position": { "x": 100, "y": 200 },
      "data": { "model": "gpt-4" }
    }
  ],
  "edges": [
    {
      "id": "edge_1",
      "source": "input_1",
      "target": "llm_1"
    }
  ]
}
```

Connection Management

- Automatic port typing validation
- Connection persistence using CRDTs for conflict resolution

4. Properties Panel

HTML Mockup Features: Component-specific settings, validation indicators

Frontend Implementation

```
// PropertiesPanel.tsx
const PropertiesPanel = ({ selectedComponent }) => {
  const [formData, setFormData] = useState(selectedComponent.config);

  const handleSubmit = () => {
    api.updateComponentConfig(selectedComponent.id, formData);
  };

  return (
    <Formik
      initialValues={selectedComponent.config}
      onSubmit={handleSubmit}
    >
      {({ values }) => (
        <Form>
          <Field name="temperature" as={RangeInput} min={0} max={1} step={0.1} />
          <Field name="apiKey" as={SecretInput} />
          <ValidationIndicator isValid={validateConfig(values)} />
        </Form>
      )}
    </Formik>
  );
};
```

Backend Services

- **PATCH /api/components/{id}**: Updates component configuration

```
{
  "config": {
    "temperature": 0.7,
    "maxTokens": 500,
    "stream": true
  }
}
```

Security Features

- Secret encryption using AWS KMS
- Config validation against JSON Schema

5. Deployment & Preview

HTML Mockup Features: Environment selector, deployment logs, preview pane

Frontend Implementation

```
// DeploymentPanel.tsx
const DeploymentPanel = () => {
  const [env, setEnv] = useState('staging');

  return (
    <div className="space-y-4">
      <EnvironmentSelector
        environments={['local', 'staging', 'prod']}
        onSelect={setEnv}
      />
      <PreviewPane env={env} />
      <DeploymentLogs />
      <Button
        onClick={() => api.deployToEnvironment(env)}
        variant="primary"
      >
        Deploy to {env}
      </Button>
    </div>
  );
};
```

Backend Services

- **POST /api/deploy/preview**: Generates preview environment
- **POST /api/deploy/production**: Production deployment pipeline

```
sequenceDiagram
  Frontend->>API: POST /deploy/preview
  API->>Kubernetes: Create Namespace
  Kubernetes-->>API: Deployment Status
  API->>Frontend: WebSocket Updates
```

Technical Standards

1. **Version Control**: All components use strict SemVer with Renovate bot for updates
2. **API Contracts**: OpenAPI 3.0 specifications with automated client generation
3. **Error Handling**: Unified error codes across frontend/backend

```
enum ErrorCodes {
  COMPONENT_VERSION_MISMATCH = 'CVM-409',
  INVALID_CONNECTION_TYPE = 'ICT-422',
  DEPLOYMENT_QUOTA_EXCEEDED = 'DQE-429'
}
```

4. **Performance**: Canvas interactions optimized with WebWorkers
5. **Access Control**: RBAC implemented at component/API level

This architecture enables safe composition of AI components while maintaining enterprise-grade reliability and scalability. The system is designed for 1000+ concurrent users with sub-200ms response times for canvas operations.

⁂