OptiView Pro

Industrial Architecture & Implementation Guide

Software Architecture Handout

Version 1.0 – November 24, 2025

Contents

1 Introduction

OptiView Pro is an interactive decision-support dashboard for multi-objective optimization (MOO). Its core UX philosophy is *Linked Dual-Space Understanding*: correlating design inputs (*Design Space*) with performance outcomes (*Objective Space*) to enable intuitive trade-off exploration and confident selection of Pareto-optimal solutions.

The customer requires a full-stack Python implementation with Streamlit, integrating an existing PyTorch model (either as a live object or a serialized file) plus feature-space ranges as inputs.

2 Scope and Goals

2.1 Primary User Goals

- Identify Pareto-optimal solutions quickly.
- Understand trade-offs across multiple objectives.
- Link causes (inputs) to effects (objectives and ML predictions).
- Filter and compare candidate solutions to make data-driven final choices.

2.2 In-Scope Features

- Linked dashboard with four panels + right sidebar controls.
- Design Space 2D prediction heatmap with decision boundary.
- Objective Space interactive 3D scatter with Pareto emphasis and optional ghost cloud.
- Parallel Coordinates for multi-dimensional filtering.
- Inspector for exact values of hovered/selected points.
- Two-way hover and selection linking, with optional visual linking line toggle.
- Support for either precomputed MOO datasets or on-the-fly objective evaluation.

2.3 Out-of-Scope (v1)

- Automated optimization algorithms (assumed to be run externally).
- Collaborative real-time multi-user editing (read-only multi-user allowed).
- Custom physics simulators beyond supplied objective hooks.

3 Functional Requirements

3.1 Dashboard Panels

1. **Design Space (Top-Left)**:
    - Render X1–X2 mesh grid heatmap showing $P(\text{Class A})$ from PyTorch model.
    - Overlay sampled design points.
    - Display decision boundary at probability threshold $p = 0.5$ (dashed contour).
    - Hover a point to highlight corresponding point in Objective Space.
    - Box-brush selection filters/dims points in Objective Space and Parallel Coordinates.

2. **Objective Space (Top-Right)**:
   - Interactive 3D scatter for ObjA, ObjB, ObjC.
   - Pareto points emphasized by size/brightness.
   - Discrete objective encoded by color/glyph.
   - Optional ghosted "all possible points" context cloud with opacity control.
   - Hover and selection linked bidirectionally with Design Space.
3. **Parallel Coordinates (Bottom-Left)**:
   - Axes for all 4 objectives (3 continuous + 1 discrete).
   - Range brushing (or sidebar sliders fallback) updates global selection.
4. **Inspector (Bottom-Right)**:
   - Table showing exact input values, objective values, and discrete class.
   - Shows hovered point and currently selected set.
5. **Right Sidebar Controls**:
   - Toggle linking line on/off.
   - Discrete objective filter: *Combined / Material A / Material B.*
   - Show/hide ghost cloud and opacity slider.
   - Mesh resolution and sampling controls.

3.2 Data Inputs

- **Model Input**:
  - PyTorch model object OR serialized model file (.pt/.pth).
  - Binary classification output (probability of class=1).
- **Feature Space Ranges**:
  - At minimum ranges for X1 and X2.
  - Optional additional dimensions for sampling if provided.
- **Optional Dataset Upload**:
  - CSV/Parquet with precomputed samples and objectives.

4 Non-Functional Requirements

4.1 Performance

- Grid inference for heatmap must complete within interactive latency (target < 1s for 200×200 grid on CPU).
- Dashboard remains responsive with up to $N = 20{,}000$ samples; cloud downsample to $\leq 5{,}000$ for 3D rendering.
- Use caching for model and derived data.

4.2 Reliability

- Graceful handling of missing/invalid files.
- Deterministic Pareto computation given objective arrays.
- Reproducible sampling via stored random seeds.

4.3 Security

- Uploaded models are treated as **trusted artifacts only**. Torch serialization uses pickle and can execute arbitrary code; for untrusted environments accept only TorchScript/ONNX or `state_dict` weights.
- Limit file size, enforce extension whitelist, and validate schema.
- Run behind HTTPS reverse proxy in production.

4.4 Maintainability

- Clean separation into Presentation / Application / Domain / Infrastructure layers.
- Pluggable interfaces for model predictors and objective evaluators.
- Full unit-test coverage of Domain and Application layers.

5 System Architecture Overview

5.1 High-Level Layers

1. **Presentation Layer (UI)**: Streamlit layout, Plotly panels, sidebar, theming.
2. **Application Layer**: Coordinators managing session state, cross-filtering, caching, and orchestration.
3. **Domain Layer**: Sampling, objective evaluation, Pareto sorting, and data schema.
4. **Infrastructure Layer**: Model loading, persistence (local/S3/DB), logging, deployment.

5.2 Component Diagram (Textual)

- **Streamlit App**: coordinates UI panels and reads/writes global state.
- **State Store**: single source of truth for hover/selection/filtering/settings.
- **Model Adapter**: normalizes inference for live or file-based PyTorch models.
- **Sampler**: builds mesh grid and/or point samples from feature ranges.
- **Objective Evaluator**: customer-provided function converting inputs to objectives.
- **Pareto Engine**: computes non-dominated solutions.
- **Data Source**: loads optional precomputed datasets.

5.3 Data Flow

1. Load model and feature ranges.
2. Build mesh grid and compute model probabilities.
3. Load or sample design points.
4. Evaluate objectives (or read from dataset).
5. Compute Pareto mask.
6. Render panels.
7. User hover/selection updates State Store, re-renders linked views.

6 Module Design

6.1 Repository Structure

```
optiview_pro/
|-- app.py
|-- requirements.txt
|-- core/
|   |-- model_adapter.py
|   |-- data_schema.py
|   |-- sampling.py
|   |-- objectives.py
|   |-- pareto.py
|   |-- filters.py
|   +-- state.py
|-- ui/
|   |-- sidebar.py
|   |-- panels/
|   |   |-- design_space.py
|   |   |-- objective_space.py
|   |   |-- parallel_coords.py
|   |   +-- inspector.py
|   +-- theme.py
+-- utils/
    |-- caching.py
    |-- perf.py
    +-- logging.py
```

6.2 Core Interfaces

6.2.1 Model Predictor Protocol

```python
from typing import Protocol
import numpy as np

class IModelPredictor(Protocol):
    def predict_proba(self, x: np.ndarray) -> np.ndarray:
        """Return probability of class=1 for each row in x."""
```

6.2.2 Objective Evaluator Protocol

```
from typing import Protocol, Dict
import numpy as np

class IObjectiveEvaluator(Protocol):
    def evaluate(self, x: np.ndarray, proba: np.ndarray) -> Dict[str, np.
        ndarray]:
        """
        Return dict with keys: ObjA, ObjB, ObjC, Discrete.
        Each value is shape (N,).
        """
```

6.2.3 DataFrame Contract

All downstream UI assumes a unified DataFrame:

$$\texttt{df\_points}: \{id, X1, X2, ObjA, ObjB, ObjC, Discrete, is\_pareto\}$$

6.3 State Management

6.3.1 State Store

```
from dataclasses import dataclass, field
from typing import Set, Optional, Dict, Any

@dataclass
class Filters:
    selected_ids: Set[int] = field(default_factory=set)
    hover_id: Optional[int] = None
    discrete_filter: str = "Combined"
    obj_ranges: Dict[str, Any] = field(default_factory=dict)

@dataclass
class UISettings:
    show_link: bool = False
    show_cloud: bool = True
    cloud_opacity: float = 0.08
    mesh_res: int = 200
    sample_n: int = 2000

@dataclass
class AppState:
    df_points: Any = None
    grid: Dict[str, Any] = field(default_factory=dict)
    filters: Filters = field(default_factory=Filters)
    ui: UISettings = field(default_factory=UISettings)
```

6.3.2 Session State Wiring

- Instantiate AppState once and store at `st.session_state["state"]`.
- Panels read from and write to this object only.
- Any update triggers a re-render of all panels (Streamlit reactive model).

7 Implementation Guide

7.1 Model Adapter

```python
import torch
import numpy as np

class ModelAdapter:
    def __init__(self, model=None, model_path=None, device="cpu"):
        if model is None and model_path is None:
            raise ValueError("Provide model or model_path.")
        self.device = device
        self.model = model or torch.load(model_path, map_location=device)
        self.model.eval()

    @torch.no_grad()
    def predict_proba(self, x: np.ndarray) -> np.ndarray:
        xt = torch.from_numpy(x).float().to(self.device)
        logits = self.model(xt)

        if logits.shape[-1] == 1:
            proba = torch.sigmoid(logits).squeeze(-1)
        else:
            proba = torch.softmax(logits, dim=-1)[:, 1]

        return proba.detach().cpu().numpy()
```

7.2 Mesh Grid + Heatmap

```python
import numpy as np

def make_mesh(x1_range, x2_range, res):
    x1 = np.linspace(*x1_range, res)
    x2 = np.linspace(*x2_range, res)
    xx, yy = np.meshgrid(x1, x2)
    grid_points = np.c_[xx.ravel(), yy.ravel()]
    return xx, yy, grid_points
```

7.3 Sampling Design Points

```python
def sample_points(ranges, n, seed=0):
    rng = np.random.default_rng(seed)
    X = np.zeros((n, len(ranges)))
    for j, (lo, hi) in enumerate(ranges):
        X[:, j] = rng.uniform(lo, hi, size=n)
    return X
```

7.4 Objective Evaluation Hook

This is customer-owned code.

```python
class ObjectiveEvaluator:
    def evaluate(self, x, proba):
        # TODO: replace with customer computation
        ObjA = ...
        ObjB = ...
        ObjC = ...
        Discrete = (proba >= 0.5).astype(int)
        return dict(ObjA=ObjA, ObjB=ObjB, ObjC=ObjC, Discrete=Discrete)
```

7.5 Pareto Front Computation

```
import numpy as np

def pareto_mask(F):
    N = F.shape[0]
    is_p = np.ones(N, dtype=bool)
    for i in range(N):
        if not is_p[i]:
            continue
        dom = np.all(F <= F[i], axis=1) & np.any(F < F[i], axis=1)
        dom[i] = False
        is_p[dom] = False
    return is_p
```

7.6 UI Panels

7.6.1 Layout

```
import streamlit as st
from ui.sidebar import render_sidebar
from ui.panels.design_space import render_design_space
from ui.panels.objective_space import render_objective_space
from ui.panels.parallel_coords import render_parallel_coords
from ui.panels.inspector import render_inspector

st.set_page_config(layout="wide")
render_sidebar()

tl, tr = st.columns(2)
bl, br = st.columns(2)

with tl: render_design_space()
with tr: render_objective_space()
with bl: render_parallel_coords()
with br: render_inspector()
```

7.6.2 Design Space (2D Heatmap + Brushing)

```
import plotly.graph_objects as go
from streamlit_plotly_events import plotly_events

def render_design_space():
    s = st.session_state["state"]
    df, grid = s.df_points, s.grid

    fig = go.Figure()
    fig.add_trace(go.Heatmap(
        x=grid["xx"][0], y=grid["yy"][:,0],
        z=grid["prob"].reshape(grid["xx"].shape),
        colorbar=dict(title="P(Class A)")
    ))
    fig.add_trace(go.Contour(
        x=grid["xx"][0], y=grid["yy"][:,0],
        z=grid["prob"].reshape(grid["xx"].shape),
        contours=dict(start=0.5, end=0.5, size=1),
        showscale=False,
        line=dict(dash="dash", width=2),
        name="Decision Boundary"
    ))

    selected = s.filters.selected_ids
    colors = ["rgba(255,255,255,0.9)" if i in selected
              else "rgba(255,255,255,0.3)" for i in df["id"]]

    fig.add_trace(go.Scatter(
        x=df["X1"], y=df["X2"], mode="markers",
        marker=dict(size=6, color=colors),
        customdata=df["id"], name="Samples"
    ))

    fig.update_layout(height=420, dragmode="select")
    events = plotly_events(fig, hover_event=True, select_event=True,
                           key="design_events")

    if events:
        e = events[-1]
        if "customdata" in e:
            s.filters.hover_id = int(e["customdata"])
        if "selected_points" in e:
            ids = {int(df.iloc[i]["id"]) for i in e["selected_points"]}
            s.filters.selected_ids = ids

    st.plotly_chart(fig, use_container_width=True)
```

7.6.3 Objective Space (3D Scatter + Cloud)

```
import plotly.graph_objects as go
from streamlit_plotly_events import plotly_events

def render_objective_space():
    s = st.session_state["state"]
    df = s.df_points

    # Discrete filter
    f = s.filters.discrete_filter
    if f == "Material A Only":
        dfv = df[df["Discrete"] == 0]
    elif f == "Material B Only":
        dfv = df[df["Discrete"] == 1]
    else:
        dfv = df

    pareto = dfv[dfv["is_pareto"]]
    cloud  = dfv[~dfv["is_pareto"]]

    fig = go.Figure()
    if s.ui.show_cloud:
        fig.add_trace(go.Scatter3d(
            x=cloud["ObjA"], y=cloud["ObjB"], z=cloud["ObjC"],
            mode="markers",
            marker=dict(size=2, opacity=s.ui.cloud_opacity, color="grey"),
            customdata=cloud["id"], name="Cloud"
        ))
    fig.add_trace(go.Scatter3d(
        x=pareto["ObjA"], y=pareto["ObjB"], z=pareto["ObjC"],
        mode="markers",
        marker=dict(size=6, opacity=0.95, color=pareto["Discrete"]),
        customdata=pareto["id"], name="Pareto Front"
    ))

    fig.update_layout(height=420, scene=dict(
        xaxis_title="ObjA", yaxis_title="ObjB", zaxis_title="ObjC"
    ))

    events = plotly_events(fig, hover_event=True, select_event=True,
                           key="objective_events")

    if events:
        e = events[-1]
        if "customdata" in e:
            s.filters.hover_id = int(e["customdata"])
        if "selected_points" in e:
            ids = {int(dfv.iloc[i]["id"]) for i in e["selected_points"]}
            s.filters.selected_ids = ids

    st.plotly_chart(fig, use_container_width=True)
```

7.6.4 Parallel Coordinates

```
import plotly.graph_objects as go

def render_parallel_coords():
    s = st.session_state["state"]
    df = s.df_points

    fig = go.Figure(go.Parcoords(
        line=dict(color=df["is_pareto"].astype(int), showscale=False),
        dimensions=[
            dict(label="ObjA", values=df["ObjA"]),
            dict(label="ObjB", values=df["ObjB"]),
            dict(label="ObjC", values=df["ObjC"]),
            dict(label="Discrete", values=df["Discrete"])
        ]
    ))
    fig.update_layout(height=360)
    st.plotly_chart(fig, use_container_width=True)

    # Industrial fallback for range filtering:
    # add sidebar sliders bound to s.filters.obj_ranges
```

7.6.5 Inspector

```
def render_inspector():
    s = st.session_state["state"]
    df = s.df_points
    hid = s.filters.hover_id
    sel = s.filters.selected_ids
    st.subheader("Details & Inspector")
    if hid is not None:
        st.markdown("**Hovered Point**")
        st.dataframe(df[df["id"] == hid])

    if sel:
        st.markdown("**Selected Points**")
        st.dataframe(df[df["id"].isin(sel)])
```

7.7 Sidebar Controls

```
def render_sidebar():
    s = st.session_state["state"]

    st.sidebar.header("Controls & Filters")
    s.ui.show_link = st.sidebar.toggle("Show Linking Line", value=s.ui.
       show_link)
    s.ui.show_cloud = st.sidebar.toggle("Show All Possible Points Cloud",
       value=s.ui.show_cloud)
    s.ui.cloud_opacity = st.sidebar.slider("Feasibility Cloud Opacity",
                                           0.0, 0.2, s.ui.cloud_opacity)

    s.filters.discrete_filter = st.sidebar.radio(
        "Discrete Objective Filter",
        ["Combined", "Material A Only", "Material B Only"],
        index=["Combined","Material A Only","Material B Only"].index(s.
           filters.discrete_filter)
    )

    s.ui.mesh_res = st.sidebar.slider("Mesh Resolution", 50, 400, s.ui.
       mesh_res)
    s.ui.sample_n = st.sidebar.slider("Sample Size", 100, 20000, s.ui.
       sample_n)

    if st.sidebar.button("Recompute"):
        st.session_state["recompute_flag"] = True
```

8 Caching and Performance Engineering

8.1 Caching Strategy

- `st.cache_resource`: model loading.
- `st.cache_data`: mesh inference, objective evaluation, Pareto mask.
- Cache keys incorporate: model hash, mesh_res, sample_n, seed, objective config.

8.2 Downsampling Strategy

- If $N_{\text{cloud}} > 50k$, sample uniformly to $5k$ for 3D panel.
- Preserve all Pareto points (never downsample Pareto).

8.3 GPU Optionality

If CUDA is available, allow user to enable GPU inference via device toggle. Always batch grid inference to avoid memory spikes.

9 Persistence, Export, and Reproducibility

9.1 Session Export

Provide a download button for:

- Full dataset with Pareto flag.
- Selected subset only.
- JSON snapshot of filters + UI settings.

9.2 Reproducibility

Store the following in any exported snapshot:

- Random seed used for sampling.
- Model identifier/hash.
- Objective-evaluator version string.
- Feature ranges and mesh resolution.

10 Testing Strategy

10.1 Unit Tests

- Pareto correctness (known dominated/non-dominated sets).
- Objective evaluator contract (shape/type checks).
- Sampling bounds tests.
- Model adapter probability range $\in [0, 1]$.

10.2 Integration Tests

- End-to-end pipeline: model $\rightarrow$ grid $\rightarrow$ objectives $\rightarrow$ Pareto $\rightarrow$ DataFrame.
- Regression snapshot tests for DataFrame schema.

10.3 UI Smoke Tests

Automated Playwright tests:

- Page loads without error.
- Sidebar recompute triggers.
- Plot panels render non-empty traces.

11 Deployment

11.1 Docker

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8501
CMD ["streamlit", "run", "app.py",
     "--server.address=0.0.0.0", "--server.port=8501"]
```

11.2 Production Topology

- Nginx reverse proxy for TLS termination and gzip compression.
- Horizontal scaling via container replicas (stateless UI).
- Shared persistence (S3/DB) for datasets and exports.

11.3 Observability

- Structured logs (JSON) for model load, inference times, recompute events.
- Optional Prometheus metrics endpoint for compute latency, active sessions.

12 Roadmap

12.1 v1.1 Enhancements

- Custom Streamlit component for true parcoords brushing events.
- Advanced Pareto layers (epsilon dominance, crowding distance).
- Saved "analysis workspaces" for quick reopening.

12.2 v2.0 Scalability

- Optional FastAPI compute backend + job queue for heavy simulations.
- Multi-user collaboration (shared read-only sessions).
- Role-based access control.

13 Conclusion

This handout provides a complete industrial-grade blueprint to implement OptiView Pro in Streamlit with pluggable PyTorch model inference, objective evaluation, Pareto computation, and deeply linked interactive visual analytics.