Python Code

```python
import pandas as pd
import numpy as np
from collections import defaultdict
from fastapi import FastAPI, UploadFile, File, Form, HTTPException
import tempfile, os
from scipy.stats import skew, kurtosis, entropy, normaltest, jarque_bera
from sklearn.feature_selection import mutual_info_classif, mutual_info_regression

app = FastAPI(
    title="Scalable Statistical EDA Engine | Phase 1–3",
    debug=True
)

MAX_SAMPLE_ROWS = 50_000
MAX_MI_ROWS = 20_000
DISTANCE_WIN_MARGIN = 0.25
RF_PRIOR_BONUS = 0.20
BOOSTING_PRIOR_BONUS = 0.18


 #---------------- Safety net for JSON Conversion & Policy Helpers ------------
def distance_model_eligible(signals):
    return (
        signals.get("categorical_ratio", 0.0) < 0.3
        and signals.get("sparsity_ratio", 1.0) < 0.35
        and signals.get("missing_ratio", 0.0) < 0.05
        and signals.get("p_over_n", 1.0) < 1.0
        and signals.get("n_features", 0) <= 25
        and signals.get("n_samples", 0) <= 50_000
    )

def random_forest_preferred(signals):
    return (
        signals.get("nonlinear_signal_strength", 0.0) >= 0.25
        and signals.get("n_samples", 0) >= 300
        and signals.get("n_features", 0) <= 200
        and signals.get("sparsity_ratio", 0.0) <= 0.5
        and signals.get("categorical_ratio", 0.0) <= 0.4
    )


def _safe_float(val):
    if val is None:
        return None
    if not isinstance(val, (int, float, np.integer, np.floating)):
        return None
    if np.isnan(val) or np.isinf(val):
        return None
    return float(val)

```

```python


def to_json_safe(obj):
    if isinstance(obj, (np.integer,)):
        return int(obj)
    if isinstance(obj, (np.floating,)):
        return _safe_float(float(obj))
    if isinstance(obj, (np.bool_,)):
        return bool(obj)
    if isinstance(obj, float):
        return _safe_float(obj)
    if isinstance(obj, dict):
        return {k: to_json_safe(v) for k, v in obj.items()}
    if isinstance(obj, list):
        return [to_json_safe(v) for v in obj]
    return obj


# ------------------ Utilities ------------------

def save_temp_file(uploaded_file: UploadFile) -> str:
    suffix = os.path.splitext(uploaded_file.filename)[1]
    with tempfile.NamedTemporaryFile(delete=False, suffix=suffix) as tmp:
        tmp.write(uploaded_file.file.read())
    return tmp.name




def smart_sample(df, max_rows):
    sampled = df if len(df) <= max_rows else df.sample(max_rows, random_state=42)
    return smart_fillna(sampled)

def smart_fillna(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()

    # Drop columns that are entirely NaN
    df = df.dropna(axis=1, how="all")

    for col in df.columns:
        if pd.api.types.is_numeric_dtype(df[col]):
            median = df[col].median()
            if np.isnan(median):
                df[col] = df[col].fillna(0.0)
            else:
                df[col] = df[col].fillna(median)
        else:
            mode = df[col].mode()
            if len(mode) > 0:
                df[col] = df[col].fillna(mode.iloc[0])
            else:
                df[col] = df[col].fillna("UNKNOWN")

    return df
```

```python


# ------------------ Step 1 ------------------

def step1_dataset_overview(csv_path):
    df = pd.read_csv(csv_path, nrows=MAX_SAMPLE_ROWS)

    return {
        "rows_sampled": len(df),
        "columns": int(df.shape[1]),
        "memory_mb_estimate": round(df.memory_usage(deep=True).sum() / (1024 ** 2),
2),
        "column_types": {c: str(t) for c, t in df.dtypes.items()},
        "sample_rows": df.head(5).to_dict(orient="records")
    }, df


# ------------------ Step 2 ------------------

def step2_target_statistics(series):
    s = series.dropna()

    if len(s) == 0:
        raise ValueError("Target column contains only missing values")

    stats = {
        "unique_values": int(s.nunique()),
        "unique_ratio": round(s.nunique() / max(len(s), 1), 4),
        "entropy": _safe_float(float(entropy(s.value_counts(normalize=True))))
    }

    if pd.api.types.is_numeric_dtype(s):
        stats.update({
            "skewness": _safe_float(float(skew(s))),
            "kurtosis": _safe_float(float(kurtosis(s))),
            "outlier_ratio_proxy": _safe_float(float(
                ((s < s.quantile(0.01)) | (s > s.quantile(0.99))).mean()
            ))
        })
    else:
        stats["max_class_ratio"] = float(s.value_counts(normalize=True).max())

    return stats


# ------------------ Step 3 ------------------

def step3_problem_inference(target_stats):
    if target_stats["unique_ratio"] <= 0.05 or target_stats["unique_values"] <= 20:
        return {"problem_type": "classification", "confidence": 0.95}
    return {"problem_type": "regression", "confidence": 0.95}


# ------------------ Step 4 (MOST IMPORTANT FIXES) ------------------
```

```python

def step4_feature_target_signals(df, target_col, task_type):
    df = smart_sample(df, MAX_MI_ROWS)

    X = df.drop(columns=[target_col])
    y = df[target_col].dropna()
    X = X.loc[y.index]
    n_samples = int(len(X))
    n_features = int(X.shape[1])

    numeric_X = X.select_dtypes(include="number")
    categorical_X = X.select_dtypes(exclude="number")
    missing_ratio = float(X.isna().mean().mean()) if X.shape[1] > 0 else 0.0
    p_over_n = float(X.shape[1] / max(len(X), 1))

    # No usable numeric features
    if numeric_X.shape[1] == 0 and categorical_X.shape[1] == 0:
        return {
            "linear_signal_strength": 0.0,
            "nonlinear_signal_strength": 0.0,
            "sparsity_ratio": 1.0,
            "categorical_ratio": 0.0,
            "note": "no_usable_features"
        }

    y_enc = y if task_type == "regression" else y.astype("category").cat.codes

    # Constant target → MI & corr invalid
    if y_enc.nunique() <= 1:
        return {
            "linear_signal_strength": 0.0,
            "nonlinear_signal_strength": 0.0,
            "sparsity_ratio": float((numeric_X == 0).mean().mean()),
            "categorical_ratio": float(categorical_X.shape[1] / max(X.shape[1], 1)),
            "note": "constant_target"
        }

    if numeric_X.shape[1] > 0:
        corr = numeric_X.corrwith(y_enc).abs()
        linear_signal = _safe_float(float(corr.mean()))
        if linear_signal is None:
            linear_signal = 0.0
        skew_mean = _safe_float(float(numeric_X.skew(numeric_only=True).abs().mean()))
        kurt_mean =
_safe_float(float(numeric_X.kurtosis(numeric_only=True).abs().mean()))
        skew_kurtosis_score = float((skew_mean or 0.0) + (kurt_mean or 0.0))
    else:
        linear_signal = 0.0
        skew_kurtosis_score = 0.0

    # Build mixed feature matrix for MI
    if categorical_X.shape[1] > 0:
        cat_encoded = categorical_X.apply(lambda col: pd.factorize(col)[0])
        X_mi = pd.concat([numeric_X, cat_encoded], axis=1)
```

```
        discrete_mask = [False] * numeric_X.shape[1] + [True] * cat_encoded.shape[1]
    else:
        X_mi = numeric_X
        discrete_mask = [False] * numeric_X.shape[1]

    if X_mi.shape[1] > 0:
        mi = (
            mutual_info_regression(X_mi, y_enc, discrete_features=discrete_mask)
            if task_type == "regression"
            else mutual_info_classif(X_mi, y_enc, discrete_features=discrete_mask)
        )
        mi_mean = _safe_float(float(np.mean(mi)))
        if mi_mean is None:
            mi_mean = 0.0
    else:
        mi_mean = 0.0

    nonlinearity_index = (
        float(mi_mean / max(linear_signal, 1e-6))
        if linear_signal > 0.0
        else float(mi_mean)
    )

    return {
        "linear_signal_strength": linear_signal,
        "nonlinear_signal_strength": mi_mean,
        "sparsity_ratio": float((numeric_X == 0).mean().mean()) if numeric_X.shape[1]
> 0 else 1.0,
        "categorical_ratio": float(categorical_X.shape[1] / max(X.shape[1], 1)),
        "nonlinearity_index": _safe_float(float(nonlinearity_index)),
        "missing_ratio": missing_ratio,
        "p_over_n": p_over_n,
        "skew_kurtosis_score": _safe_float(float(skew_kurtosis_score)),
        "n_samples": n_samples,
        "n_features": n_features
    }



# ------------------ Step 5 ------------------

def step5_model_family_scoring(task_type, signals):
    cat_ratio = signals.get("categorical_ratio", 0.0)
    nonlin_idx = signals.get("nonlinearity_index", 0.0) or 0.0
    sparsity = signals.get("sparsity_ratio", 0.0)
    linear = signals.get("linear_signal_strength", 0.0)
    nonlinear = signals.get("nonlinear_signal_strength", 0.0)
    missing_ratio = signals.get("missing_ratio", 0.0)
    p_over_n = signals.get("p_over_n", 0.0)
    n_samples = signals.get("n_samples", 0)
    skew_kurtosis = signals.get("skew_kurtosis_score", 0.0)
    # Distance models are sensitive to high p/n, sparsity, missingness, and large n.
    size_penalty = 0.0
    if n_samples >= 100_000:
```

```
        size_penalty = 0.4
    elif n_samples >= 50_000:
        size_penalty = 0.2

    distance_score = (
        (1.0 - sparsity)
        - cat_ratio * 0.25
        - missing_ratio * 0.2
        - max(p_over_n - 1.0, 0.0) * 0.2
        - size_penalty
    )
    if not distance_model_eligible(signals):
        distance_score = -1.0

    tree_boost = 0.0
    if nonlin_idx and nonlin_idx > 1.2:
        tree_boost += 0.1
    if missing_ratio >= 0.1:
        tree_boost += 0.05
    if cat_ratio >= 0.3:
        tree_boost += 0.05

    scores = {
        "linear_models": (
            linear
            - sparsity * 0.3
            - cat_ratio * 0.2
            - max(p_over_n - 1.0, 0.0) * 0.1
            - skew_kurtosis * 0.1
        ),
        "tree_based": (
            nonlinear
            + cat_ratio * 0.2
            + missing_ratio * 0.2
            + max(nonlin_idx - 1.0, 0.0) * 0.1
            + tree_boost
        ),
        "distance_based": distance_score
    }

    if task_type == "classification":
        scores["probabilistic"] = 0.7 + cat_ratio * 0.2 + max(p_over_n - 1.0, 0.0) *
0.1

    for k in list(scores.keys()):
        scores[k] = max(min(float(scores[k]), 1.0), -1.0)

    return scores


# ------------------ Step 6 ------------------

def step6_final_family_selection(scores):
    if not scores:
```

```
        return {"chosen_family": None, "reason": "no_valid_models"}

    best = max(scores, key=scores.get)
    return {
        "chosen_family": best,
        "generalization_score": round(max(min(float(scores[best]), 1.0), -1.0), 4),
        "rule": "highest_statistical_signal"
    }


def step6b_generalization_selection(scores, signals):
    if not scores:
        return {"chosen_family": None, "reason": "no_valid_models"}

    n_samples = signals.get("n_samples", 0)
    n_features = signals.get("n_features", 0)
    sample_ratio = float(n_samples / max(n_features, 1))
    tail_score = float(signals.get("skew_kurtosis_score") or 0.0)

    variance_risk = {
        "linear_models": 0.05,
        "tree_based": 0.25,
        "distance_based": 0.55,
        "probabilistic": 0.10
    }

    def sample_stress(family):
        if family == "distance_based":
            return max(0.0, 1.0 - np.log(max(sample_ratio, 1e-6)) / 4.0)
        if family == "tree_based":
            return max(0.0, 1.0 - np.log(max(sample_ratio, 1e-6)) / 6.0)
        return 0.0

    def instability(family):
        if family == "distance_based":
            return min(0.3, tail_score / 20.0)
        if family == "tree_based":
            return min(0.15, tail_score / 30.0)
        if family == "linear_models":
            return min(0.1, tail_score / 40.0)
        return min(0.1, tail_score / 40.0)

    def distance_reliability(signals):
        score = 1.0
        nonlin = signals.get("nonlinearity_index", 0.0) or 0.0
        score -= min(0.4, nonlin / 2)
        score -= min(0.3, (signals.get("skew_kurtosis_score", 0.0) or 0.0) / 20)
        score -= min(0.2, np.log(signals.get("n_features", 1) + 1) / 4)
        if nonlin > 1.2:
            score -= 0.25
        elif nonlin > 0.8:
            score -= 0.15
        n = signals.get("n_samples", 0)
        if n > 10_000:
```

```
            score -= 0.20
        if n > 25_000:
            score -= 0.35
        return max(0.0, score)

    family_prior = {
        "tree_based": 0.10,
        "linear_models": 0.05,
        "distance_based": -0.15
    }

    robustness_bonus = {
        "linear_models": 0.15,
        "tree_based": 0.10,
        "distance_based": 0.00,
        "probabilistic": 0.05
    }

    generalization_scores = {}
    for family, s_signal in scores.items():
        s_adj = float(s_signal)

        if family == "distance_based":
            s_adj *= distance_reliability(signals)

        g = (
            0.45 * s_adj
            - 0.20 * variance_risk.get(family, 0.15)
            - 0.15 * sample_stress(family)
            - 0.10 * instability(family)
            + 0.10 * robustness_bonus.get(family, 0.05)
            + family_prior.get(family, 0.0)
        )

        # ✅ FIX 1: probabilistic models collapse under nonlinear interactions
        if family == "probabilistic":
            if signals.get("nonlinearity_index", 0.0) >= 0.8:
                g -= 0.25

        # existing RF preference
        if family == "tree_based" and random_forest_preferred(signals):
            g += RF_PRIOR_BONUS

        # ✅ FIX 2: reward trees for real interaction capture
        if family == "tree_based" and signals.get("nonlinearity_index", 0.0) >= 1.0:
            g += 0.20

        generalization_scores[family] = max(min(float(g), 1.0), -1.0)

    distance = generalization_scores.get("distance_based", -1.0)
    tree = generalization_scores.get("tree_based", -1.0)
    linear = generalization_scores.get("linear_models", -1.0)

    if distance >= 0:
```

```
        if not (
            distance > tree + DISTANCE_WIN_MARGIN
            and distance > linear + DISTANCE_WIN_MARGIN
        ):
            generalization_scores["distance_based"] = max(
                min(distance - 0.3, 1.0), -1.0
            )

    if (
        "distance_based" in generalization_scores
        and "tree_based" in generalization_scores
    ):
        if (
            generalization_scores["distance_based"]
            - generalization_scores["tree_based"]
            < 0.15
        ):
            best = "tree_based"
        else:
            best = max(generalization_scores, key=generalization_scores.get)
    else:
        best = max(generalization_scores, key=generalization_scores.get)

    return {
        "chosen_family": best,
        "generalization_score": round(float(generalization_scores[best]), 4),
        "rule": "multi_signal_generalization_score",
        "generalization_scores": {
            k: round(float(v), 4) for k, v in generalization_scores.items()
        }
    }



# ------------------ Step 7 ------------------

def step7_family_risks(signals):
    return {
        "linear_models": "risk" if signals["linear_signal_strength"] <
signals["nonlinear_signal_strength"] else "low",
        "tree_based": "risk" if signals["linear_signal_strength"] > 0.7 else "low",
        "distance_based": "risk" if signals["sparsity_ratio"] > 0.4 else "low"
    }


# ------------------ Step 8 ------------------

def step8_hyperparameter_hints(df):
    rows, cols = df.shape
    hints = {}

    if rows > 100_000:
        hints["tree_depth"] = "limit"
    if cols > 100:
```

```
        hints["feature_sampling"] = "enable"

    return hints


# ------------------ Step 9 ------------------

def step9_family_algorithms(task_type):
    families = {
        "linear_models": [
            "linear_regression",
            "ridge",
            "lasso",
            "elastic_net",
            "logistic_regression",
            "linear_svm"
        ],
        "tree_based": [
            "decision_tree",
            "random_forest",
            "extra_trees",
            "gradient_boosting",
            "xgboost_or_lightgbm"
        ],
        "distance_based": [
            "knn",
            "svm_rbf"
        ],
        "probabilistic": [
            "naive_bayes",
            "gaussian_process"
        ]
    }

    if task_type == "regression":
        families["probabilistic"] = ["bayesian_ridge", "gaussian_process"]
    if task_type != "classification":
        families["linear_models"] = [
            "linear_regression",
            "ridge",
            "lasso",
            "elastic_net"
        ]

    return families


# ------------------ Step 10 ------------------

def step10_algorithm_tests(df, target_col, task_type, signals):
    df = smart_sample(df, MAX_MI_ROWS)
    y = df[target_col].dropna()
    X = df.drop(columns=[target_col]).loc[y.index]
    numeric_X = X.select_dtypes(include="number")
```

```python
    categorical_X = X.select_dtypes(exclude="number")

    tests = {}

    # Target distribution tests (for linear assumptions)
    if pd.api.types.is_numeric_dtype(y):
        y_vals = y.astype(float).values
        if len(y_vals) >= 8:
            tests["target_normality_pvalue"] = _safe_float(float(normaltest(y_vals).pvalue))
            tests["target_jarque_bera_pvalue"] = _safe_float(float(jarque_bera(y_vals).pvalue))
        else:
            tests["target_normality_pvalue"] = None
            tests["target_jarque_bera_pvalue"] = None

    # Feature-level diagnostics
    if numeric_X.shape[1] > 0:
        corr = numeric_X.corrwith(y).abs()
        tests["top_linear_corr"] = _safe_float(float(corr.max())) if len(corr) else None
        tests["mean_linear_corr"] = _safe_float(float(corr.mean())) if len(corr) else None
        tests["zero_variance_features"] = int((numeric_X.nunique() <= 1).sum())
        tests["high_sparsity_features"] = int(((numeric_X == 0).mean() > 0.8).sum())
    else:
        tests["top_linear_corr"] = None
        tests["mean_linear_corr"] = None
        tests["zero_variance_features"] = 0
        tests["high_sparsity_features"] = 0
    tests["categorical_features"] = int(categorical_X.shape[1])

    # Algo-specific guidance
    tests["knn_scale_required"] = True if numeric_X.shape[1] > 0 else False
    tests["svm_rbf_recommended"] = True if signals["nonlinear_signal_strength"] > signals["linear_signal_strength"] else False
    tests["linear_models_recommended"] = True if signals["linear_signal_strength"] > 0.2 else False
    tests["tree_models_recommended"] = True if signals["nonlinear_signal_strength"] > 0.2 else False
    tests["distance_models_recommended"] = bool(
        signals.get("categorical_ratio", 0.0) < 0.3
        and signals.get("sparsity_ratio", 1.0) < 0.4
        and signals.get("p_over_n", 1.0) < 1.0
        and signals.get("n_samples", 0) <= 50_000
    )

    if task_type == "classification":
        tests["class_imbalance_ratio"] = _safe_float(float(y.value_counts(normalize=True).max()))

        # ---------------- Tree / Boosting Diagnostics ----------------

    # Interaction proxy: MI vs correlation gap
```

```
    interaction_strength = None
    if (
        tests.get("mean_linear_corr") is not None
        and signals.get("nonlinear_signal_strength") is not None
    ):
        interaction_strength = max(
            0.0,
            signals["nonlinear_signal_strength"] - tests["mean_linear_corr"]
        )

    # Noise proxy (lower is better for boosting)
    noise_proxy = None
    if tests.get("top_linear_corr") is not None:
        noise_proxy = 1.0 - tests["top_linear_corr"]

    tests["tree_interaction_strength"] = _safe_float(interaction_strength)
    tests["noise_proxy"] = _safe_float(noise_proxy)

    # Random Forest suitability
    tests["random_forest_suitable"] = bool(
        interaction_strength is not None
        and interaction_strength >= 0.15
        and signals.get("n_samples", 0) >= 300
        and signals.get("sparsity_ratio", 0.0) <= 0.5
    )

    # Gradient Boosting suitability
    tests["gradient_boosting_suitable"] = bool(
        interaction_strength is not None
        and interaction_strength >= 0.20
        and noise_proxy is not None
        and noise_proxy <= 0.6
        and signals.get("n_samples", 0) >= 1_000
        and signals.get("sparsity_ratio", 0.0) <= 0.4
    )

    # XGBoost / LightGBM suitability
    tests["xgboost_lightgbm_suitable"] = bool(
        signals.get("nonlinear_signal_strength", 0.0) >= 0.30
        and (
            signals.get("skew_kurtosis_score", 0.0) >= 1.5
            or signals.get("missing_ratio", 0.0) >= 0.1
        )
        and signals.get("n_samples", 0) >= 2_000
    )

    return tests


# ------------------ Step 11 ------------------
def step11_boosting_needed(df, target_col, task_type, signals, generalization_scores):
    rows, cols = df.shape

    nonlin = signals.get("nonlinear_signal_strength", 0.0)
```

```
    linear = signals.get("linear_signal_strength", 0.0)
    sparsity = signals.get("sparsity_ratio", 0.0)
    skew_kurt = signals.get("skew_kurtosis_score", 0.0)
    n = signals.get("n_samples", 0)

    tree_score = generalization_scores.get("tree_based", -1.0)
    linear_score = generalization_scores.get("linear_models", -1.0)

    boosting_recommended = (
        tree_score >= 0.15
        and tree_score > linear_score + 0.10
        and nonlin >= 0.3
        and sparsity <= 0.4
        and skew_kurt >= 1.5
        and n >= 1_000
        and n <= 200_000
    )

    return {
        "boosting_recommended": bool(boosting_recommended),
        "preferred_boosting_model": (
            "xgboost_or_lightgbm" if boosting_recommended else None
        ),
        "rule":
"tree_generalization_high_and_structured_nonlinearity_and_sufficient_data",
        "signals": {
            "tree_generalization_score": tree_score,
            "linear_generalization_score": linear_score,
            "nonlinear_signal_strength": nonlin,
            "skew_kurtosis_score": skew_kurt,
            "rows": n,
            "cols": cols
        }
    }


# ------------------ Step 12 ------------------

def step12_top_algorithms_min_latency(task_type, signals, scores):
    """
    Exposes candidate algorithms per family with a latency-aware bias,
    but does NOT prematurely suppress Random Forest or Boosting when
    the data clearly supports them.
    """

    top = {}

    # ---------------- Linear family (always cheap, always baseline) ----------------
    if task_type == "classification":
        top["linear_models"] = ["logistic_regression", "linear_svm"]
    else:
        top["linear_models"] = ["linear_regression", "ridge"]

    # ---------------- Tree family (progressive exposure) ----------------
```

```
    tree_algos = ["decision_tree", "extra_trees"]

    # Prefer Random Forest when structure + data size justify it
    if random_forest_preferred(signals):
        tree_algos.append("random_forest")

    # Allow boosting only when signal + scale justify it
    if (
        signals.get("nonlinear_signal_strength", 0.0) >= 0.30
        and signals.get("n_samples", 0) >= 1_000
        and signals.get("sparsity_ratio", 0.0) <= 0.4
    ):
        tree_algos.extend([
            "gradient_boosting",
            "xgboost_or_lightgbm"
        ])

    top["tree_based"] = tree_algos

    # ---------------- Distance family (strict gating) ----------------
    distance_ok = (
        scores.get("distance_based", 0.0) >= 0.25
        and signals.get("categorical_ratio", 0.0) < 0.3
        and signals.get("sparsity_ratio", 1.0) < 0.5
        and signals.get("p_over_n", 1.0) < 1.0
        and signals.get("n_samples", 0) <= 50_000
        and signals.get("nonlinearity_index", 0.0) <= 1.0
    )

    top["distance_based"] = ["knn", "svm_rbf"] if distance_ok else []

    # ---------------- Probabilistic (baseline, not dominant) ----------------
    if task_type == "classification":
        top["probabilistic"] = ["naive_bayes", "gaussian_process"]
    else:
        top["probabilistic"] = ["bayesian_ridge", "gaussian_process"]

    # ---------------- Family ranking (used later by Step 13 / 15) ----------------
    family_rank = sorted(scores.items(), key=lambda kv: kv[1], reverse=True)
    ordered_families = [f for f, _ in family_rank]

    return {
        "ordered_families": ordered_families,
        "top_algorithms_by_family": top,
        "latency_bias": "favor_simple_models_before_boosting"
    }



# ------------------ Step 13 ------------------

def step13_top5_algorithms(step12, best_family):
    MAX_PER_FAMILY = 2
    ordered = step12["ordered_families"]
```

```
    by_family = step12["top_algorithms_by_family"]
    top5 = []
    family_counts = defaultdict(int)
    for fam in ordered:
        for algo in by_family.get(fam, []):
            if family_counts[fam] >= MAX_PER_FAMILY:
                continue
            if algo not in top5:
                top5.append(algo)
                family_counts[fam] += 1
            if len(top5) >= 5:
                break
        if len(top5) >= 5:
            break

    if best_family == "distance_based":
        tree_algos = by_family.get("tree_based", [])
        has_tree = any(a in tree_algos for a in top5)
        if not has_tree and tree_algos:
            # Ensure at least one tree-based model is present
            if len(top5) >= 5:
                top5.pop()
            top5.append(tree_algos[0])

    return {
        "top_5_algorithms": top5,
        "selection_rule": "family_rank_then_latency_bias"
    }


# ------------------ Step 15 ------------------

def step15_overall_best5_models(step12, generalization_scores, algo_tests, signals,
boosting_info):
    by_family = step12["top_algorithms_by_family"]

    ranked = []

    # ---- Build scored candidate list ----
    for fam, fam_score in generalization_scores.items():
        if fam_score < -0.1:
            continue

        for algo in by_family.get(fam, []):
            bonus = 0.0

            # Model-specific suitability bonuses
            if algo == "random_forest" and random_forest_preferred(signals):
                bonus += 0.20

            if algo == "gradient_boosting" and
algo_tests.get("gradient_boosting_suitable"):
                bonus += 0.25

```

```
            if algo == "xgboost_or_lightgbm" and algo_tests.get("xgboost_lightgbm_suitable"):
                bonus += 0.30

            ranked.append((algo, fam, fam_score + bonus))

    # Sort by adjusted score
    ranked.sort(key=lambda x: x[2], reverse=True)

    # ---- Enforce family diversity (THIS WAS MISSING) ----
    MAX_PER_FAMILY = 2
    family_counts = {}
    final = []

    for algo, fam, score in ranked:
        family_counts.setdefault(fam, 0)

        if family_counts[fam] >= MAX_PER_FAMILY:
            continue

        final.append((algo, fam, score))
        family_counts[fam] += 1

        if len(final) >= 5:
            break

    return {
        "overall_best_5_models": [
            {
                "algorithm": a,
                "family": f,
                "generalization_score": round(float(s), 4),
                "selection_reason": (
                    "preferred_tree_model"
                    if a == "random_forest" and random_forest_preferred(signals)
                    else "boosting_recommended"
                    if a in ["gradient_boosting", "xgboost_or_lightgbm"]
                    and boosting_info.get("boosting_recommended")
                    else "baseline_candidate"
                )
            }
            for a, f, s in final
        ],
        "selection_rule": "generalization_score_rank_with_family_diversity"
    }



# ------------------ Step 14 ------------------

def step14_condition_based_recommendations(df, target_col, task_type, signals, algo_tests):
    rows, cols = df.shape
    n = rows
```

```
    p = max(cols - 1, 0)
    cat_ratio = signals.get("categorical_ratio", 0.0)
    sparsity = signals.get("sparsity_ratio", 0.0)
    linear = signals.get("linear_signal_strength", 0.0)
    nonlinear = signals.get("nonlinear_signal_strength", 0.0)

    recs = []

    def add_rule(condition, models, rationale):
        if condition:
            recs.append({
                "recommended_models": models,
                "rationale": rationale
            })

    add_rule(
        n < 1_000 and p <= 50,
        ["logistic_regression", "linear_regression", "naive_bayes",
"decision_tree_small", "knn"],
        "Very small dataset; prefer simple models to reduce overfitting."
    )

    add_rule(
        n <= 200_000 and p <= 200,
        ["random_forest", "xgboost_or_lightgbm", "ridge"],
        "Moderate size; trees capture interactions, GLM as baseline."
    )

    add_rule(
        n > 200_000,
        ["sgd_linear", "linear_svm", "lightgbm_hist"],
        "Very large dataset; prioritize scalable models."
    )

    add_rule(
        p > n * 2,
        ["lasso", "elastic_net", "feature_selection_plus_glm"],
        "High-dimensional (p >> n); regularized linear models or feature selection."
    )

    add_rule(
        sparsity >= 0.6,
        ["linear_svm_sparse", "logistic_regression_sparse", "naive_bayes"],
        "High sparsity; linear or NB models are robust."
    )

    add_rule(
        cat_ratio >= 0.4,
        ["catboost", "lightgbm", "random_forest", "target_encoding_plus_glm"],
        "Many categorical features; tree boosting handles categorical splits."
    )
    add_rule(
        signals.get("missing_ratio", 0.0) >= 0.2,
        ["lightgbm", "xgboost_or_lightgbm", "random_forest"],
```

```
        "High missingness; tree-based models handle missing values better."
    )
    add_rule(
        signals.get("skew_kurtosis_score", 0.0) >= 2.0,
        ["random_forest", "xgboost_or_lightgbm", "robust_regression_huber"],
        "Heavy tails/outliers; prefer trees or robust regression."
    )

    add_rule(
        task_type == "regression" and linear >= 0.3 and nonlinear <= 0.2,
        ["ridge", "lasso", "elastic_net"],
        "Continuous target with linear signal; linear models are sufficient."
    )

    add_rule(
        task_type == "regression" and nonlinear > linear + 0.1,
        ["random_forest", "xgboost_or_lightgbm", "svm_rbf"],
        "Continuous target with nonlinear signal; tree ensembles or kernels perform
better."
    )

    add_rule(
        task_type == "classification",
        ["logistic_regression", "random_forest", "xgboost_or_lightgbm"],
        "Classification: start with GLM + trees + SVM."
    )

    if task_type == "classification":
        imb = algo_tests.get("class_imbalance_ratio")
        add_rule(
            imb is not None and imb >= 0.8,
            ["xgboost_weighted", "random_forest_weighted",
"weighted_logistic_regression"],
            "Imbalanced classes; use class-weighted models or boosting with weights."
        )

    add_rule(
        algo_tests.get("high_sparsity_features", 0) >= max(1, int(p * 0.3)),
        ["linear_svm_sparse", "logistic_regression_sparse"],
        "Many sparse features; sparse linear models perform well."
    )
    add_rule(
        algo_tests.get("distance_models_recommended", False),
        ["knn", "svm_linear"],
        "Low dimensional, mostly numeric, and low sparsity; distance models can work
well."
    )

    return {
        "rules_triggered": recs,
        "note": "Rules are heuristics; use as guidance with validation."
    }


```

```python
# ------------------ API ------------------

@app.post("/eda/phase1_3")
def run_optimized_eda(
    file: UploadFile = File(...),
    target_column: str = Form(...)
):
    csv_path = None

    try:
        csv_path = save_temp_file(file)
        step1, df = step1_dataset_overview(csv_path)

        if target_column not in df.columns:
            raise HTTPException(
                status_code=400,
                detail=f"Target column '{target_column}' not found"
            )

        step2 = step2_target_statistics(df[target_column])
        step3 = step3_problem_inference(step2)
        step4 = step4_feature_target_signals(df, target_column, step3["problem_type"])
        step5 = step5_model_family_scoring(step3["problem_type"], step4)
        step6 = step6b_generalization_selection(step5, step4)
        step7 = step7_family_risks(step4)
        step8 = step8_hyperparameter_hints(df)
        step9 = step9_family_algorithms(step3["problem_type"])
        step10 = step10_algorithm_tests(df, target_column, step3["problem_type"], step4)
        step11 = step11_boosting_needed(
                df,
                target_column,
                step3["problem_type"],
                step4,
                step6["generalization_scores"]
            )

        step12 = step12_top_algorithms_min_latency(step3["problem_type"], step4, step5)
        step13 = step13_top5_algorithms(step12, step6["chosen_family"])
        step14 = step14_condition_based_recommendations(df, target_column, step3["problem_type"], step4, step10)
        step15 = step15_overall_best5_models(
                step12=step12,
                generalization_scores=step6["generalization_scores"],
                algo_tests=step10,
                signals=step4,
                boosting_info=step11
            )




        response ={
```

```
            "STEP_1_DATASET_OVERVIEW": step1,
            "STEP_2_TARGET_STATISTICS": step2,
            "STEP_3_PROBLEM_INFERENCE": step3,
            "STEP_4_FEATURE_TARGET_SIGNALS": step4,
            "STEP_5_MODEL_FAMILY_SCORING": step5,
            "STEP_6_FAMILY_RISK_DIAGNOSTICS": step7,
            "STEP_7_HYPERPARAMETER_HINTS": step8,
            "STEP_8_FINAL_MODEL_SELECTION": step6,
            "distance_models_status": {
                "eligible": distance_model_eligible(step4),
                "confidence": "conditional",
                "conditions": [
                    "low dimensional",
                    "mostly numeric",
                    "low sparsity",
                    "no heavy skew/kurtosis"
                ]
            },
            "STEP_9_FAMILY_ALGORITHMS": step9,
            "STEP_10_ALGO_SPECIFIC_TESTS": step10,
            "STEP_11_BOOSTING_RECOMMENDATION": step11,
            "STEP_12_TOP_ALGOS_MIN_LATENCY": step12,
            "STEP_13_TOP_5_ALGOS": step13,
            "STEP_14_CONDITION_BASED_RECOMMENDATIONS": step14,
            "STEP_15_OVERALL_BEST_5_MODELS": step15
        }

        return to_json_safe(response)

    finally:
        if csv_path and os.path.exists(csv_path):
            os.remove(csv_path)

```