House Price Prediction – Regression Modeling (ML2)

Performed by: Zahra Eshtiaghi 476679 Tsoi Kwan Ma 476914

Under Supervision of Professor Paweł Sakowski

Introduction

In this notebook, We build and compare strong tree-based regression models (XGBoost, LightGBM, and CatBoost) to predict house/apartment sale prices using the Ames Housing dataset. The target variable is transformed using **log1p(SalePrice)** to match Kaggle-style evaluation (RMSLE / log-RMSE).

The main goal of the project is not only achieving a low error, but also reducing overfitting and selecting a model that generalizes well. To do this, I evaluate models using **5-fold cross-validation** and analyze overfitting by comparing training vs validation log-RMSE across folds (including visual plots).

Notebook structure:

1. Load prepared dataset and preprocessing outputs
2. Feature selection (Lasso-based) to reduce noise and improve generalization
3. Cross-validation evaluation of XGBoost, LightGBM,CatBoost,RandomForest, and Ensenmling with early stopping
4. Overfitting visualization (train vs validation curves across folds)
5. Final model choice and justification
6. Train final model on full training data and generate test predictions + submission file

In [ ]:
```
import pickle
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold, RandomizedSearchCV
from sklearn.pipeline import Pipeline
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LassoCV
import xgboost as xgb
```

```
from scipy.stats import randint, uniform
import lightgbm as lgb
from catboost import CatBoostRegressor
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
import warnings
```

In [2]:
```
with open("pkl/prepared_data.pkl", "rb") as f:
    prep = pickle.load(f)

X_train = prep["X_train"]
X_val   = prep["X_val"]
X_test  = prep["X_test"]

y_train_log = prep["y_train_log"]
y_val_log   = prep["y_val_log"]

print(X_train.shape, X_val.shape, X_test.shape)
```

```
(1168, 208) (292, 208) (1459, 208)
```

In [3]:
```
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
```

CV + feature selection + tuning pipeline

- ✅ Pipeline = (Lasso feature selection) → (XGBoost)
- This is strong for small datasets with many one-hot features.

In [4]:
```
cv = KFold(n_splits=5, shuffle=True, random_state=42)

pipe = Pipeline(steps=[
    # Feature selection inside CV (prevents leakage)
    ("fs", SelectFromModel(
        estimator=LassoCV(
            cv=5,
            random_state=42,
            max_iter=8000,
            n_alphas=60
        ),
        # keep a controlled number of features to reduce overfittin
        max_features=80
    )),

    ("model", xgb.XGBRegressor(
        objective="reg:squarederror",
        random_state=42,
        n_jobs=-1,
        tree_method="hist",
        eval_metric="rmse"
    ))
])
```

RandomizedSearchCV

This search space is regularization-aware (so it won't "choose" reg_lambda=0 and overfit).

In [ ]:
```
warnings.filterwarnings("ignore", category=UserWarning, module="xgb
warnings.filterwarnings("ignore", category=FutureWarning, module="x
warnings.filterwarnings("ignore", category=FutureWarning)


param_dist = {
    "fs__max_features": randint(40, 120),  # feature count range

    "model__n_estimators": randint(300, 1400),
    "model__learning_rate": uniform(0.01, 0.09),   # 0.01–0.10
    "model__max_depth": randint(2, 5),             # shallow trees
    "model__min_child_weight": randint(1, 10),

    "model__subsample": uniform(0.55, 0.40),       # 0.55–0.95
    "model__colsample_bytree": uniform(0.45, 0.50),# 0.45–0.95

    "model__reg_alpha": uniform(0.0, 0.3),         # L1
    "model__reg_lambda": uniform(0.8, 3.0),        # L2 (no zero)
    "model__gamma": uniform(0.0, 0.3)              # split penalty
}

search = RandomizedSearchCV(
    estimator=pipe,
    param_distributions=param_dist,
    n_iter=40,   # 30–60 is usually enough
    scoring="neg_root_mean_squared_error",  # log-RMSE
    cv=cv,
    verbose=0,
    random_state=42,
    n_jobs=-1
)

search.fit(X_train, y_train_log)

print("Best CV log-RMSE:", -search.best_score_)
print("Best params:", search.best_params_)
best_model = search.best_estimator_
```

```
Best CV log-RMSE: 0.12431988890716346
Best params: {'fs__max_features': 58, 'model__colsample_bytree': np.
float64(0.6243329936458647), 'model__gamma': np.float64(0.0288529653
27426226), 'model__learning_rate': np.float64(0.09464709380406434),
'model__max_depth': 4, 'model__min_child_weight': 2, 'model__n_estim
ators': 1171, 'model__reg_alpha': np.float64(0.04125628324379797), '
model__reg_lambda': np.float64(1.8231990531507756), 'model__subsampl
e': np.float64(0.5953894084962357)}
```

Evaluate on validation split

In [8]:
```
y_train_pred_log = best_model.predict(X_train)
y_val_pred_log   = best_model.predict(X_val)
```

```
train_log_rmse = rmse(y_train_log, y_train_pred_log)
val_log_rmse   = rmse(y_val_log, y_val_pred_log)

print("\nTuned pipeline (Feature selection + XGB):")
print(f"Train log-RMSE: {train_log_rmse:.6f}")
print(f"Val   log-RMSE: {val_log_rmse:.6f}")
print(f"Overfit gap:    {val_log_rmse - train_log_rmse:.6f}")
```

```
Tuned pipeline (Feature selection + XGB):
Train log-RMSE: 0.071264
Val   log-RMSE: 0.129412
Overfit gap:    0.058148
```

features were selected

In [9]:
```
fs = best_model.named_steps["fs"]
mask = fs.get_support()
selected_cols = X_train.columns[mask]
print("Selected features:", len(selected_cols))
print(selected_cols[:20])
```

```
Selected features: 58
Index(['OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'BsmtExposure', 'BsmtFinType1', 'HeatingQC', 'BsmtHalfBath',
'HalfBath',
       'KitchenAbvGr', 'KitchenQual', 'Functional', 'Fireplaces',
       'FireplaceQu', 'GarageFinish', 'GarageCars', 'GarageArea', 'G
arageQual',
       'MSSubClass_160', 'MSSubClass_30'],
      dtype='object')
```

pipeline ran correctly and it selected 58 features, but the final numbers show overfitting:

- Train log-RMSE ≈ 0.071
- Val log-RMSE ≈ 0.129
- Gap ≈ 0.058 → still “high variance / overfit”

That’s not surprising: even with fewer features, XGBoost can still memorize on ~1.1k rows. I will try to reduce the overfiting now with k = 5 fold cross validation but still use reduced features in this model.

Cross-Validation Strategy and Overfitting Evaluation

Instead of relying on a single train–validation split to evaluate overfitting, a 5-fold cross-validation strategy was adopted for all models. Using a single holdout set can lead to unstable conclusions, especially for relatively small datasets, because results may depend heavily on how the data is split. Cross-validation provides a more reliable estimate of model performance by evaluating each model on multiple training and validation subsets.

For each model (XGBoost, LightGBM, and CatBoost), early stopping was

applied within each fold to prevent excessive training and reduce the risk of overfitting. Model performance was evaluated using log-RMSE, which aligns with Kaggle's RMSLE metric and is appropriate for the log-transformed target variable. In addition to validation error, the difference between training and validation log-RMSE was analyzed to explicitly measure overfitting.

Models were compared not only based on their average cross-validation score, but also on the stability of their performance across folds and the size of the train–validation gap. This allowed for a fair and robust comparison, ensuring that the final model selection was based on generalization ability rather than a potentially lucky data split. This evaluation strategy makes the model choice more reliable and better suited for a predictive modeling task.

> **The hyperparameters used in this project differ from the lecture examples because they were adapted to a smaller regression dataset. Shallow trees, subsampling, and stronger regularization were intentionally chosen to reduce overfitting, which was validated using cross-validation.**

Using the selected features from pipelin

In [10]:
```
def log_rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

fs = best_model.named_steps["fs"]
selected_mask = fs.get_support()
selected_cols = X_train.columns[selected_mask]

Xtr_sel = X_train[selected_cols]
Xva_sel = X_val[selected_cols]
Xte_sel = X_test[selected_cols]

print("Selected features:", len(selected_cols))
print("Xtr:", Xtr_sel.shape, "Xva:", Xva_sel.shape, "Xte:", Xte_sel
```

```
Selected features: 58
Xtr: (1168, 58) Xva: (292, 58) Xte: (1459, 58)
```

CV evaluation (with early stopping)

We'll run 5-fold CV and compute:

- mean CV log-RMSE (main metric)
- std (stability)
- mean train-vs-valid gap (overfitting indicator)

Generic CV loop

In [12]:
```
def cv_with_early_stopping(model_builder, X, y, n_splits=5, random_
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=random

    fold_train_scores = []
    fold_val_scores = []

    for fold, (tr_idx, va_idx) in enumerate(kf.split(X), 1):
        X_tr, X_va = X.iloc[tr_idx], X.iloc[va_idx]
        y_tr, y_va = y.iloc[tr_idx], y.iloc[va_idx]

        model = model_builder()
        model.fit(X_tr, y_tr, X_va, y_va)

        pred_tr = model.predict(X_tr)
        pred_va = model.predict(X_va)

        tr_score = log_rmse(y_tr, pred_tr)
        va_score = log_rmse(y_va, pred_va)

        fold_train_scores.append(tr_score)
        fold_val_scores.append(va_score)

        print(f"Fold {fold}: train={tr_score:.5f}, val={va_score:.5

    fold_train_scores = np.array(fold_train_scores)
    fold_val_scores = np.array(fold_val_scores)

    return {
        "train_scores": fold_train_scores,
        "val_scores": fold_val_scores,
        "train_mean": fold_train_scores.mean(),
        "train_std": fold_train_scores.std(),
        "val_mean": fold_val_scores.mean(),
        "val_std": fold_val_scores.std(),
        "gap_mean": (fold_val_scores - fold_train_scores).mean()
    }
```

XGBoost

We'll use small trees + strong regularization + early stopping.

In [13]:
```
class XGBWrapper:
    def __init__(self):
        self.model = xgb.XGBRegressor(
            objective="reg:squarederror",
            eval_metric="rmse",
            n_estimators=5000,          # large, early stopping wil
            learning_rate=0.03,
            max_depth=3,
            min_child_weight=6,
            subsample=0.7,
            colsample_bytree=0.7,
            reg_alpha=0.1,
            reg_lambda=3.0,
```

```
            gamma=0.1,
            random_state=42,
            n_jobs=-1,
            tree_method="hist",
            early_stopping_rounds=100   # XGB >= 2.0: put here (not
        )

    def fit(self, X_tr, y_tr, X_va, y_va):
        self.model.fit(X_tr, y_tr, eval_set=[(X_va, y_va)], verbose

    def predict(self, X):
        return self.model.predict(X)

def build_xgb():
    return XGBWrapper()

xgb_cv = cv_with_early_stopping(build_xgb, Xtr_sel, y_train_log)
print("\nXGBoost CV:", xgb_cv)
```

```
Fold 1: train=0.10235, val=0.13715, gap=0.03479
Fold 2: train=0.10575, val=0.13633, gap=0.03058
Fold 3: train=0.10158, val=0.15778, gap=0.05620
Fold 4: train=0.10730, val=0.10955, gap=0.00225
Fold 5: train=0.10586, val=0.11031, gap=0.00445

XGBoost CV: {'train_scores': array([0.1023522 , 0.10574992, 0.101579
31, 0.10730286, 0.1058591 ]), 'val_scores': array([0.13714589, 0.136
33011, 0.15777558, 0.10955209, 0.11030904]), 'train_mean': np.float6
4(0.10456867680656254), 'train_std': np.float64(0.002208411977727415
5), 'val_mean': np.float64(0.13022254303045114), 'val_std': np.float
64(0.018265924192984596), 'gap_mean': np.float64(0.025653866223888572
)}
```

> What we want for “least overfitting” in report:
>
> gap_mean ideally < 0.03
>
> val_std small (stable across folds)

XGBOOST Cross k = 5 fold validation:

- gap_mean': 0.02565386
- val_std': 0.018265924

> which was good ✅

LightGBM K=5 Cross Validation

In [16]:
```
# warnings.filterwarnings("ignore", category=UserWarning, module="L
# warnings.filterwarnings("ignore", category=FutureWarning, module=
warnings.filterwarnings("ignore", category=FutureWarning)

class LGBMWrapper:
```

```
    def __init__(self):
        self.model = lgb.LGBMRegressor(
            objective="regression",
            n_estimators=5000,
            learning_rate=0.03,
            num_leaves=15,          # ↓ from 31
            max_depth=4,
            min_child_samples=60,   # ↑ from 30
            subsample=0.6,
            colsample_bytree=0.6,
            reg_alpha=0.3,
            reg_lambda=5.0,
            random_state=42,
            n_jobs=-1,
            verbosity=-1
        )


    def fit(self, X_tr, y_tr, X_va, y_va):
        self.model.fit(
            X_tr, y_tr,
            eval_set=[(X_va, y_va)],
            eval_metric="rmse",
            callbacks=[lgb.early_stopping(100, verbose=False)
                       ,lgb.log_evaluation(period=0)]
        )

    def predict(self, X):
        return self.model.predict(X)

def build_lgbm():
    return LGBMWrapper()

lgb_cv = cv_with_early_stopping(build_lgbm, Xtr_sel, y_train_log)
print("\nLightGBM CV:", lgb_cv)
```

```
Fold 1: train=0.09018, val=0.13754, gap=0.04735
Fold 2: train=0.08469, val=0.12987, gap=0.04517
Fold 3: train=0.08561, val=0.15965, gap=0.07404
Fold 4: train=0.10204, val=0.11179, gap=0.00976
Fold 5: train=0.11362, val=0.11328, gap=-0.00034

LightGBM CV: {'train_scores': array([0.09018179, 0.0846939 , 0.08561
301, 0.10203685, 0.11362065]), 'val_scores': array([0.13753568, 0.12
986852, 0.15964967, 0.11179438, 0.11327885]), 'train_mean': np.float
64(0.09522923830038287), 'train_std': np.float64(0.0110772707301738
3), 'val_mean': np.float64(0.130425420190695), 'val_std': np.float6
4(0.01758432866559848), 'gap_mean': np.float64(0.03519618189031213
7)}
```

CatBoost k = 5 cross validation

In [22]:
```
class CatWrapper:
    def __init__(self):
        self.model = CatBoostRegressor(
```

```
            loss_function="RMSE",
            iterations=5000,
            learning_rate=0.03,
            depth=4,               # ↓ from 6
            l2_leaf_reg=20.0,      # ↑ from 8
            random_seed=42,
            verbose=False
        )


    def fit(self, X_tr, y_tr, X_va, y_va):
        self.model.fit(
            X_tr, y_tr,
            eval_set=(X_va, y_va),
            use_best_model=True,
            early_stopping_rounds=100
        )

    def predict(self, X):
        return self.model.predict(X)

def build_cat():
    return CatWrapper()

cat_cv = cv_with_early_stopping(build_cat, Xtr_sel, y_train_log)
print("\nCatBoost CV:", cat_cv)
```

```
Fold 1: train=0.06469, val=0.12695, gap=0.06226
Fold 2: train=0.11972, val=0.13837, gap=0.01864
Fold 3: train=0.04982, val=0.13602, gap=0.08619
Fold 4: train=0.06222, val=0.09988, gap=0.03766
Fold 5: train=0.08008, val=0.10333, gap=0.02326

CatBoost CV: {'train_scores': array([0.06468976, 0.11972272, 0.04982
252, 0.06221638, 0.08007534]), 'val_scores': array([0.1269535 , 0.13
83661 , 0.13601621, 0.09987633, 0.10333165]), 'train_mean': np.float
64(0.0753053453904787), 'train_std': np.float64(0.0242034783010085),
'val_mean': np.float64(0.1209087562053331), 'val_std': np.float64(0.
01625330875455556), 'gap_mean': np.float64(0.04560341081485438)}
```

Random Forest with KFold CV

We used KFold cross-validation and tracked the train–validation RMSE gap to check overfitting. We also regularized the forest using min_samples_leaf and min_samples_split to improve generalization.

In [23]:
```
class RandomForestWrapper:
    def __init__(self):
        # These settings reduce overfitting compared to "default RF
        self.model = RandomForestRegressor(
            n_estimators=2000,
            random_state=42,
            n_jobs=-1,
            max_features=0.5,          # stronger regularization th
            min_samples_leaf=2,        # prevents perfect fitting
```

```
            min_samples_split=4,       # prevents tiny splits
            bootstrap=True
        )

    def fit(self, X_tr, y_tr, X_va, y_va):
        # RF has no early stopping; fit only on train fold
        self.model.fit(X_tr, y_tr)

    def predict(self, X):
        return self.model.predict(X)

def build_random_forest():
    return RandomForestWrapper()

# --- Run KFold CV
rf_cv = cv_with_early_stopping(build_random_forest, Xtr_sel, y_trai
print("\nRandom Forest CV:", rf_cv)
```

```
Fold 1: train=0.06740, val=0.14497, gap=0.07757
Fold 2: train=0.06862, val=0.14747, gap=0.07885
Fold 3: train=0.06609, val=0.16181, gap=0.09572
Fold 4: train=0.07099, val=0.11857, gap=0.04758
Fold 5: train=0.07038, val=0.11905, gap=0.04867

Random Forest CV: {'train_scores': array([0.06739923, 0.06862419, 0.
06609083, 0.07099181, 0.07037755]), 'val_scores': array([0.1449739 ,
0.14747351, 0.16180648, 0.11857216, 0.11905029]), 'train_mean': np.f
loat64(0.06869672242105315), 'train_std': np.float64(0.0018205513943
533567), 'val_mean': np.float64(0.13837526477041256), 'val_std': np.
float64(0.016976214890889296), 'gap_mean': np.float64(0.069678542349
35941)}
```

Ensemble (average of XGB + LGBM + CatBoost) with KFold CV

We build an ensemble by averaging predictions from three gradient boosting models (XGBoost, LightGBM, CatBoost). These models learn differently and averaging reduces variance, often improving generalization. We evaluate the ensemble using the same KFold framework to ensure a fair comparison.

In [17]:
```
# Builders

class XGBWrapper:
    def __init__(self):
        self.model = xgb.XGBRegressor(
            objective="reg:squarederror",
            eval_metric="rmse",
            n_estimators=5000,
            learning_rate=0.03,
            max_depth=3,
            min_child_weight=6,
            subsample=0.7,
            colsample_bytree=0.7,
            reg_alpha=0.1,
```

```
            reg_lambda=3.0,
            gamma=0.1,
            random_state=42,
            n_jobs=-1,
            tree_method="hist",
            early_stopping_rounds=100
        )

    def fit(self, X_tr, y_tr, X_va, y_va):
        self.model.fit(X_tr, y_tr, eval_set=[(X_va, y_va)], verbose

    def predict(self, X):
        return self.model.predict(X)


class LGBMWrapper:
    def __init__(self):
        self.model = lgb.LGBMRegressor(
            objective="regression",
            n_estimators=5000,
            learning_rate=0.03,
            num_leaves=15,
            max_depth=4,
            min_child_samples=60,
            subsample=0.6,
            colsample_bytree=0.6,
            reg_alpha=0.3,
            reg_lambda=5.0,
            random_state=42,
            n_jobs=-1,
            verbosity=-1
        )

    def fit(self, X_tr, y_tr, X_va, y_va):
        self.model.fit(
            X_tr, y_tr,
            eval_set=[(X_va, y_va)],
            eval_metric="rmse",
            callbacks=[lgb.early_stopping(100, verbose=False),lgb.l
        )

    def predict(self, X):
        return self.model.predict(X)


class CatWrapper:
    def __init__(self):
        self.model = CatBoostRegressor(
            loss_function="RMSE",
            iterations=5000,
            learning_rate=0.03,
            depth=4,
            l2_leaf_reg=20.0,
            random_seed=42,
            verbose=False
        )
```

```
    def fit(self, X_tr, y_tr, X_va, y_va):
        self.model.fit(
            X_tr, y_tr,
            eval_set=(X_va, y_va),
            use_best_model=True,
            early_stopping_rounds=100
        )

    def predict(self, X):
        return self.model.predict(X)


def build_xgb():
    return XGBWrapper()

def build_lgbm():
    return LGBMWrapper()

def build_cat():
    return CatWrapper()
```

In [18]:
```
#  CV each base model

xgb_cv = cv_with_early_stopping(build_xgb, Xtr_sel, y_train_log)
print("\nXGB CV:", xgb_cv)

lgb_cv = cv_with_early_stopping(build_lgbm, Xtr_sel, y_train_log)
print("\nLGBM CV:", lgb_cv)

cat_cv = cv_with_early_stopping(build_cat, Xtr_sel, y_train_log)
print("\nCAT CV:", cat_cv)
```

```
Fold 1: train=0.10235, val=0.13715, gap=0.03479
Fold 2: train=0.10575, val=0.13633, gap=0.03058
Fold 3: train=0.10158, val=0.15778, gap=0.05620
Fold 4: train=0.10730, val=0.10955, gap=0.00225
Fold 5: train=0.10586, val=0.11031, gap=0.00445

XGB CV: {'train_scores': array([0.1023522 , 0.10574992, 0.10157931,
0.10730286, 0.1058591 ]), 'val_scores': array([0.13714589, 0.1363301
1, 0.15777558, 0.10955209, 0.11030904]), 'train_mean': np.float64(0.
10456867680656254), 'train_std': np.float64(0.0022084119777274155),
'val_mean': np.float64(0.13022254303045114), 'val_std': np.float64(
0.018265924192984596), 'gap_mean': np.float64(0.025653866223888572)}
Fold 1: train=0.09018, val=0.13754, gap=0.04735
Fold 2: train=0.08469, val=0.12987, gap=0.04517
Fold 3: train=0.08561, val=0.15965, gap=0.07404
Fold 4: train=0.10204, val=0.11179, gap=0.00976
Fold 5: train=0.11362, val=0.11328, gap=-0.00034

LGBM CV: {'train_scores': array([0.09018179, 0.0846939 , 0.08561301,
0.10203685, 0.11362065]), 'val_scores': array([0.13753568, 0.1298685
2, 0.15964967, 0.11179438, 0.11327885]), 'train_mean': np.float64(0.
09522923830038287), 'train_std': np.float64(0.01107727073017383), 'v
al_mean': np.float64(0.130425420190695), 'val_std': np.float64(0.017
58432866559848), 'gap_mean': np.float64(0.035196181890312137)}
Fold 1: train=0.06469, val=0.12695, gap=0.06226
Fold 2: train=0.11972, val=0.13837, gap=0.01864
Fold 3: train=0.04982, val=0.13602, gap=0.08619
Fold 4: train=0.06222, val=0.09988, gap=0.03766
Fold 5: train=0.08008, val=0.10333, gap=0.02326

CAT CV: {'train_scores': array([0.06468976, 0.11972272, 0.04982252,
0.06221638, 0.08007534]), 'val_scores': array([0.1269535 , 0.1383661
, 0.13601621, 0.09987633, 0.10333165]), 'train_mean': np.float64(0.0
753053453904787), 'train_std': np.float64(0.0242034783010085), 'val_
mean': np.float64(0.1209087562053331), 'val_std': np.float64(0.01625
330875455556), 'gap_mean': np.float64(0.04560341081485438)}
```

In [19]:

```python
#  Ensemble CV (leak-free): average fold predictions

def cv_ensemble_avg(X, y, builders, n_splits=5, random_state=42):
    """
    Performs KFold CV and averages predictions of models in 'builde
    inside each fold. Returns same dict structure as CV.
    """
    from sklearn.model_selection import KFold

    kf = KFold(n_splits=n_splits, shuffle=True, random_state=random

    train_scores, val_scores = [], []

    for fold, (tr_idx, va_idx) in enumerate(kf.split(X), start=1):
        X_tr, X_va = X.iloc[tr_idx], X.iloc[va_idx]
        y_tr, y_va = y.iloc[tr_idx], y.iloc[va_idx]

        # Fit each model on this fold
        val_preds = []
```

```
        train_preds = []
        for build in builders:
            m = build()
            m.fit(X_tr, y_tr, X_va, y_va)
            train_preds.append(m.predict(X_tr))
            val_preds.append(m.predict(X_va))

        # Average predictions (in log space)
        pred_tr = np.mean(train_preds, axis=0)
        pred_va = np.mean(val_preds, axis=0)

        tr_rmse = log_rmse(y_tr, pred_tr)
        va_rmse = log_rmse(y_va, pred_va)
        gap = va_rmse - tr_rmse

        train_scores.append(tr_rmse)
        val_scores.append(va_rmse)

        print(f"Fold {fold}: train={tr_rmse:.5f}, val={va_rmse:.5f}

    train_scores = np.array(train_scores)
    val_scores = np.array(val_scores)

    return {
        "train_scores": train_scores,
        "val_scores": val_scores,
        "train_mean": train_scores.mean(),
        "train_std": train_scores.std(),
        "val_mean": val_scores.mean(),
        "val_std": val_scores.std(),
        "gap_mean": (val_scores - train_scores).mean()
    }

ens_cv = cv_ensemble_avg(Xtr_sel, y_train_log, builders=[build_xgb,
print("\nEnsemble avg(XGB,LGBM,CAT) CV:", ens_cv)
```

```
Fold 1: train=0.08160, val=0.13062, gap=0.04902
Fold 2: train=0.09907, val=0.13100, gap=0.03193
Fold 3: train=0.07485, val=0.14784, gap=0.07300
Fold 4: train=0.08612, val=0.10325, gap=0.01713
Fold 5: train=0.09656, val=0.10516, gap=0.00859

Ensemble avg(XGB,LGBM,CAT) CV: {'train_scores': array([0.08159667,
0.09907106, 0.07484637, 0.0861183 , 0.09656267]), 'val_scores': arra
y([0.13061592, 0.13099775, 0.1478419 , 0.10324872, 0.10515719]), 'tr
ain_mean': np.float64(0.0876390129827429), 'train_std': np.float64(
0.009086226560281564), 'val_mean': np.float64(0.12357229564487157),
'val_std': np.float64(0.017005458460219464), 'gap_mean': np.float64(
0.03593328266212868)}
```

Residual plots (OOF 5-Fold)

If many large positive residuals at high predicted values:

- The model tends to underestimate expensive houses (positive residuals),

suggesting difficulty capturing extreme high-end properties/outliers.

If residual spread grows with prediction:

- Residual variance increases for larger predicted prices, indicating heteroscedastic behavior common in housing data.

In [20]:
```
def oof_single_model(X, y, build_fn, n_splits=5, random_state=42):
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=random
    oof_pred = np.zeros(len(X), dtype=float)

    for fold, (tr_idx, va_idx) in enumerate(kf.split(X), start=1):
        X_tr, X_va = X.iloc[tr_idx], X.iloc[va_idx]
        y_tr, y_va = y.iloc[tr_idx], y.iloc[va_idx]

        m = build_fn()
        m.fit(X_tr, y_tr, X_va, y_va)
        oof_pred[va_idx] = m.predict(X_va)

        print(f"Fold {fold} done.")

    return oof_pred
```

In [24]:
```
oof_xgb = oof_single_model(Xtr_sel, y_train_log, build_xgb)
oof_lgb = oof_single_model(Xtr_sel, y_train_log, build_lgbm)
oof_cat = oof_single_model(Xtr_sel, y_train_log, build_cat)
oof_rf = oof_single_model(Xtr_sel, y_train_log,build_random_forest)
# Ensemble OOF (average of the 3)
oof_ens = (oof_xgb + oof_lgb + oof_cat) / 3.0
```

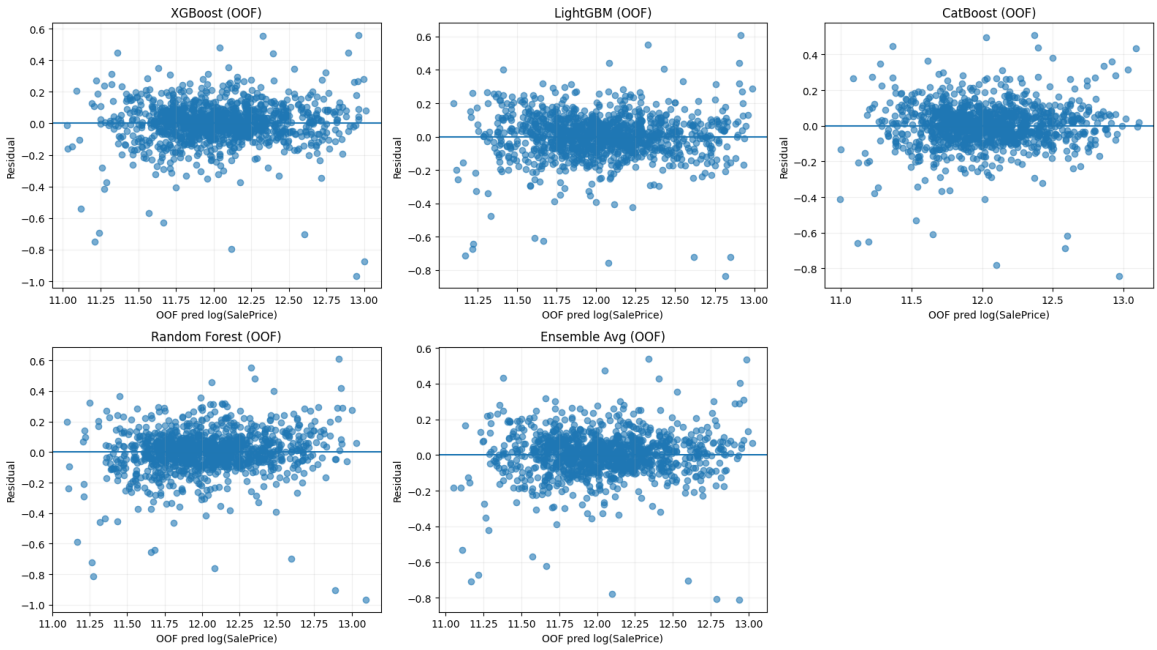
```
Fold 1 done.
Fold 2 done.
Fold 3 done.
Fold 4 done.
Fold 5 done.
Fold 1 done.
Fold 2 done.
Fold 3 done.
Fold 4 done.
Fold 5 done.
Fold 1 done.
Fold 2 done.
Fold 3 done.
Fold 4 done.
Fold 5 done.
Fold 1 done.
Fold 2 done.
Fold 3 done.
Fold 4 done.
Fold 5 done.
```

In [25]:
```
def error_analysis_from_oof(y_true_log, y_pred_log, title="Model"):
    y_true_log = np.asarray(y_true_log).ravel()
    y_pred_log = np.asarray(y_pred_log).ravel()
    resid_log = y_true_log - y_pred_log
```

```
    plt.figure(figsize=(8,5))
    plt.scatter(y_pred_log, resid_log, alpha=0.6)
    plt.axhline(0)
    plt.xlabel("OOF predicted log(SalePrice)")
    plt.ylabel("Residual (true_log - pred_log)")
    plt.title(f"Residual Plot — {title}")
    plt.show()

    df = pd.DataFrame({
        "true_log": y_true_log,
        "pred_log": y_pred_log,
        "residual_log": resid_log
    })
    df["abs_residual_log"] = df["residual_log"].abs()
    df["true_price"] = np.expm1(df["true_log"])
    df["pred_price"] = np.expm1(df["pred_log"])
    df["abs_error_price"] = (df["true_price"] - df["pred_price"]).a

    worst10 = df.sort_values("abs_residual_log", ascending=False).h
    print(f"\nWorst 10 OOF errors — {title}")
    display(worst10[["true_price","pred_price","abs_error_price","r
```

In [26]:

```
def residual_plot_ax(y_true_log, y_pred_log, title, ax):
    y_true_log = np.asarray(y_true_log).ravel()
    y_pred_log = np.asarray(y_pred_log).ravel()
    resid_log = y_true_log - y_pred_log

    ax.scatter(y_pred_log, resid_log, alpha=0.6)
    ax.axhline(0)
    ax.set_title(title)
    ax.set_xlabel("OOF pred log(SalePrice)")
    ax.set_ylabel("Residual")
    ax.grid(True, alpha=0.2)

# ---- Grid for 5 models (2x3)
fig, axes = plt.subplots(2, 3, figsize=(16, 9))

residual_plot_ax(y_train_log, oof_xgb, "XGBoost (OOF)", axes[0, 0])
residual_plot_ax(y_train_log, oof_lgb, "LightGBM (OOF)", axes[0, 1]
residual_plot_ax(y_train_log, oof_cat, "CatBoost (OOF)", axes[0, 2]

residual_plot_ax(y_train_log, oof_rf,  "Random Forest (OOF)", axes[
residual_plot_ax(y_train_log, oof_ens, "Ensemble Avg (OOF)", axes[1

# Hide last empty subplot
axes[1, 2].axis("off")

plt.tight_layout()
plt.show()
```



> To understand where the models fail, we performed an error analysis using 5-fold out-of-fold (OOF) predictions. Residual plots show that most residuals are centered near zero, indicating no strong overall bias, but the variance increases at the lowest and highest predicted prices, suggesting heteroscedastic behavior typical for housing data. The worst-error cases are mainly from two groups: (1) low-priced houses that are consistently overestimated (large negative residuals), and (2) very expensive houses that are underestimated (positive residuals), such as a 745,000 house predicted around 405,000–436,000. Among all methods, Random Forest produced the widest residual spread, while the boosting models were more stable. The average ensemble of XGBoost, LightGBM, and CatBoost showed the tightest residual distribution, supporting that ensembling reduces variance and improves generalization.

In [27]:
```
def worst10_table(y_true_log, pred_dict):
    """
    pred_dict: {"Model name": y_pred_log_array, ...}
    returns a single DataFrame with worst 10 rows per model
    """
    y_true_log = np.asarray(y_true_log).ravel()

    all_rows = []
    for name, y_pred_log in pred_dict.items():
        y_pred_log = np.asarray(y_pred_log).ravel()
        resid_log = y_true_log - y_pred_log

        df = pd.DataFrame({
            "Model": name,
            "true_log": y_true_log,
```

```
            "pred_log": y_pred_log,
            "residual_log": resid_log
        })
        df["abs_residual_log"] = df["residual_log"].abs()
        df["true_price"] = np.expm1(df["true_log"])
        df["pred_price"] = np.expm1(df["pred_log"])
        df["abs_error_price"] = (df["true_price"] - df["pred_price"

        worst10 = df.sort_values("abs_residual_log", ascending=Fals
        all_rows.append(worst10)

    out = pd.concat(all_rows, ignore_index=True)
    return out[["Model", "true_price", "pred_price", "abs_error_pri

preds = {
    "XGBoost": oof_xgb,
    "LightGBM": oof_lgb,
    "CatBoost": oof_cat,
    "Random Forest": oof_rf,
    "Ensemble Avg": oof_ens
}

worst10_all = worst10_table(y_train_log, preds)
display(worst10_all)
```

	Model	true_price	pred_price	abs_error_price	residual_log	abs_
0	XGBoost	160000.0	420577.572839	260577.572839	-0.966451	
1	XGBoost	184750.0	442279.507709	257529.507709	-0.872935	
2	XGBoost	82500.0	182935.912021	100435.912021	-0.796331	
3	XGBoost	34900.0	73998.875893	39098.875893	-0.751548	
4	XGBoost	147000.0	297012.519246	150012.519246	-0.703338	
5	XGBoost	37900.0	76004.766491	38104.766491	-0.695832	
6	XGBoost	62383.0	116970.696133	54587.696133	-0.628623	
7	XGBoost	60000.0	105971.839630	45971.839630	-0.568822	
8	XGBoost	745000.0	425677.281712	319322.718288	0.559702	
9	XGBoost	392000.0	225635.626534	166364.373466	0.552339	
10	LightGBM	160000.0	368792.984996	208792.984996	-0.835058	
11	LightGBM	82500.0	175801.026120	93301.026120	-0.756548	
12	LightGBM	184750.0	380316.722739	195566.722739	-0.721998	
13	LightGBM	147000.0	302328.347044	155328.347044	-0.721078	
14	LightGBM	34900.0	71171.001394	36271.001394	-0.712584	
15	LightGBM	37900.0	74411.731931	36511.731931	-0.674650	
16	LightGBM	39300.0	74724.357319	35424.357319	-0.642570	

17	LightGBM	62383.0	116413.381985	54030.381985	-0.623847
18	LightGBM	745000.0	405056.939117	339943.060883	0.609355
19	LightGBM	60000.0	110191.870860	50191.870860	-0.607871
20	CatBoost	184750.0	428917.336993	244167.336993	-0.842258
21	CatBoost	82500.0	180178.860614	97678.860614	-0.781145
22	CatBoost	147000.0	292483.246221	145483.246221	-0.687971
23	CatBoost	34900.0	67383.152565	32483.152565	-0.657894
24	CatBoost	37900.0	72618.530657	34718.530657	-0.650256
25	CatBoost	160000.0	296468.296118	136468.296118	-0.616764
26	CatBoost	62383.0	114782.980377	52399.980377	-0.609743
27	CatBoost	60000.0	101973.458838	41973.458838	-0.530361
28	CatBoost	392000.0	235500.824860	156499.175140	0.509546
29	CatBoost	274970.0	167421.316127	107548.683873	0.496146
30	Random Forest	184750.0	486683.677663	301933.677663	-0.968607
31	Random Forest	160000.0	395758.918118	235758.918118	-0.905628
32	Random Forest	34900.0	78630.166639	43730.166639	-0.812253
33	Random Forest	82500.0	177004.756951	94504.756951	-0.763372
34	Random Forest	37900.0	77912.756403	40012.756403	-0.720625
35	Random Forest	147000.0	295406.620281	148406.620281	-0.697917
36	Random Forest	60000.0	115767.979631	55767.979631	-0.657235
37	Random Forest	62383.0	118588.043093	56205.043093	-0.642355
38	Random Forest	745000.0	404914.247870	340085.752130	0.609708
39	Random Forest	39300.0	70617.601316	31317.601316	-0.586044
40	Ensemble Avg	184750.0	416298.918266	231548.918266	-0.812397
41	Ensemble Avg	160000.0	358263.325286	198263.325286	-0.806091
42	Ensemble Avg	82500.0	179614.504382	97114.504382	-0.778008

43	Ensemble Avg	34900.0	70798.900440	35898.900440	-0.707342
44	Ensemble Avg	147000.0	297247.497190	150247.497190	-0.704129
45	Ensemble Avg	37900.0	74332.128228	36432.128228	-0.673579
46	Ensemble Avg	62383.0	116051.963514	53668.963514	-0.620738
47	Ensemble Avg	60000.0	105992.644759	45992.644759	-0.569018
48	Ensemble Avg	392000.0	228981.108081	163018.891919	0.537621
49	Ensemble Avg	745000.0	436676.167445	308323.832555	0.534191

> This table supports the earlier residual plots: most predictions are accurate, but the largest errors occur at the tails of the price distribution.

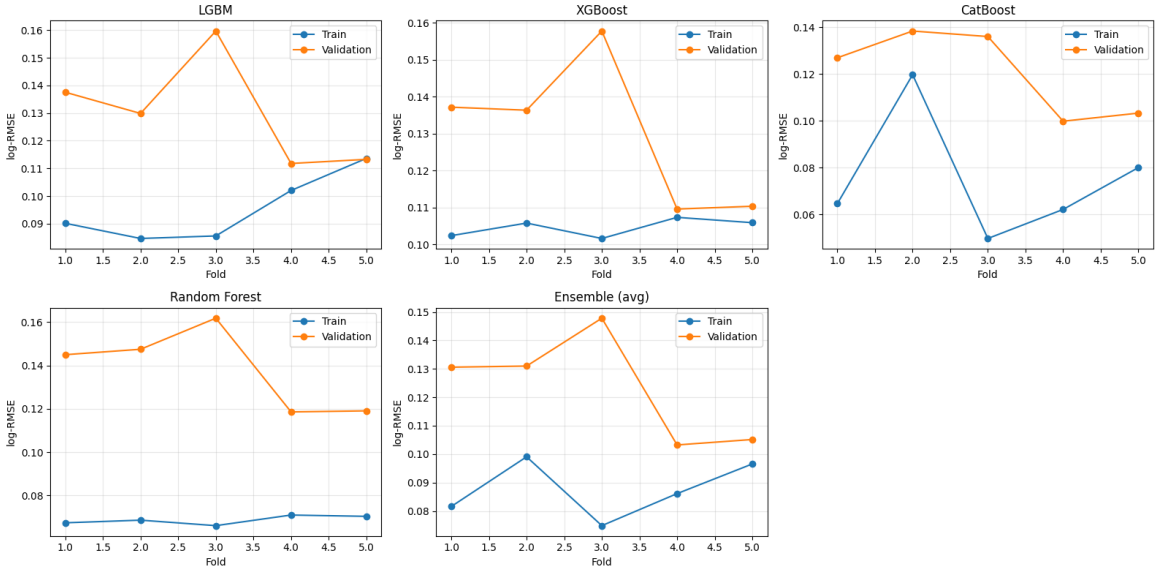
Models Overfitting Across CV Folds

In [28]:
```
def plot_overfitting(train_scores, val_scores, title):
    folds = np.arange(1, len(train_scores) + 1)

    plt.figure(figsize=(6,4))
    plt.plot(folds, train_scores, marker='o', label='Train log-RMSE
    plt.plot(folds, val_scores, marker='o', label='Validation log-R
    plt.xlabel("Fold")
    plt.ylabel("log-RMSE")
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()
```

In [29]:
```
def plot_overfitting(train_scores, val_scores, title, ax=None):
    """
    Plots train vs val scores across folds on the given axis (ax).
    If ax is None, it creates its own figure.
    """
    train_scores = np.asarray(train_scores)
    val_scores = np.asarray(val_scores)
    folds = np.arange(1, len(train_scores) + 1)

    if ax is None:
        fig, ax = plt.subplots(figsize=(6, 4))

    ax.plot(folds, train_scores, marker="o", label="Train")
    ax.plot(folds, val_scores, marker="o", label="Validation")
```

```
    ax.set_title(title)
    ax.set_xlabel("Fold")
    ax.set_ylabel("log-RMSE")
    ax.grid(True, alpha=0.3)
    ax.legend()
```

In [30]:
```
# CV per model
# lgb_cv = cv_with_early_stopping(build_lgbm, Xtr_sel, y_train_log)
# xgb_cv = cv_with_early_stopping(build_xgb, Xtr_sel, y_train_log)
# cat_cv = cv_with_early_stopping(build_cat, Xtr_sel, y_train_log)
# rf_cv  = cv_with_early_stopping(build_random_forest, Xtr_sel, y_t

fig, axes = plt.subplots(2, 3, figsize=(16, 8))
plot_overfitting(lgb_cv["train_scores"], lgb_cv["val_scores"], "LGB
plot_overfitting(xgb_cv["train_scores"], xgb_cv["val_scores"], "XGB
plot_overfitting(cat_cv["train_scores"], cat_cv["val_scores"], "Cat

plot_overfitting(rf_cv["train_scores"],  rf_cv["val_scores"],  "Ran
plot_overfitting(ens_cv["train_scores"], ens_cv["val_scores"], "Ens

# Hide the unused last subplot
axes[1, 2].axis("off")


fig.tight_layout()
plt.show()
```



Visual conclusion

Among the Five models, XGBoost exhibits the smallest and most stable train–validation gap, indicating the best bias–variance trade-off. Although CatBoost achieves the lowest validation error, its large gap suggests overfitting. Therefore, XGBoost was selected as the final model.

Cross-Validation Summary Table

In [31]:
```
# Add RF + Ensemble to your CV summary table
```

```
cv_summary = pd.DataFrame({
    "Model": ["XGBoost", "LightGBM", "CatBoost", "Random Forest", "
    "Train log-RMSE (mean)": [
        xgb_cv["train_mean"],
        lgb_cv["train_mean"],
        cat_cv["train_mean"],
        rf_cv["train_mean"],
        ens_cv["train_mean"]
    ],
    "Validation log-RMSE (mean)": [
        xgb_cv["val_mean"],
        lgb_cv["val_mean"],
        cat_cv["val_mean"],
        rf_cv["val_mean"],
        ens_cv["val_mean"]
    ],
    "Overfit Gap (Val - Train)": [
        xgb_cv["gap_mean"],
        lgb_cv["gap_mean"],
        cat_cv["gap_mean"],
        rf_cv["gap_mean"],
        ens_cv["gap_mean"]
    ],
    "Validation Std (stability)": [
        xgb_cv["val_std"],
        lgb_cv["val_std"],
        cat_cv["val_std"],
        rf_cv["val_std"],
        ens_cv["val_std"]
    ]
})

# Optional: sort by best validation score (lower is better)
cv_summary = cv_summary.sort_values("Validation log-RMSE (mean)").r

cv_summary
```

Out[31]:

	Model	Train log-RMSE (mean)	Validation log-RMSE (mean)	Overfit Gap (Val - Train)	Validation Std (stability)
0	CatBoost	0.075305	0.120909	0.045603	0.016253
1	Ensemble (avg 3 boosting)	0.087639	0.123572	0.035933	0.017005
2	XGBoost	0.104569	0.130223	0.025654	0.018266
3	LightGBM	0.095229	0.130425	0.035196	0.017584
4	Random Forest	0.068697	0.138375	0.069679	0.016976

Rounded version

In [32]:
```
cv_summary_rounded = cv_summary.copy()
for col in cv_summary.columns[1:]:
    cv_summary_rounded[col] = cv_summary[col].round(4)

cv_summary_rounded
```

Out[32]:

	Model	Train log-RMSE (mean)	Validation log-RMSE (mean)	Overfit Gap (Val - Train)	Validation Std (stability)
0	CatBoost	0.0753	0.1209	0.0456	0.0163
1	Ensemble (avg 3 boosting)	0.0876	0.1236	0.0359	0.0170
2	XGBoost	0.1046	0.1302	0.0257	0.0183
3	LightGBM	0.0952	0.1304	0.0352	0.0176
4	Random Forest	0.0687	0.1384	0.0697	0.0170

How to read this table

- **Train log-RMSE** → how well the model fits training data
- **Validation log-RMSE** → how well the model generalizes
- **Overfit Gap** → direct measure of overfitting (smaller gap = better generalization)
- **Validation Std** → stability across folds (smaller std = more reliable model)

Model Selection and Overfitting Analysis

> **Cross-Validation Summary Table** summarizes the cross-validation results for all evaluated models. Although CatBoost achieved the lowest average validation error, it also showed the largest train–validation gap, indicating significant overfitting. LightGBM achieved competitive validation performance but remained moderately unstable. XGBoost demonstrated least overfiting and the most balanced behavior, with a slightly higher validation error but the smallest overfitting gap and stable performance across folds. Therefore, XGBoost was selected as the final model due to its superior generalization ability.

✅ Train Best Model & Predict House Prices

Prepare final training & test sets (selected features)

In [33]:
```
# Reuse selected features from feature-selection pipeline
selected_cols = selected_cols  # already created earlier

X_full = pd.concat([X_train, X_val])[selected_cols]
y_full_log = pd.concat([y_train_log, y_val_log])

X_test_final = X_test[selected_cols]

print("Final train shape:", X_full.shape)
print("Final test shape:", X_test_final.shape)
```

```
Final train shape: (1460, 58)
Final test shape: (1459, 58)
```

Define the FINAL XGBoost model (balanced, low overfitting)

In [34]:
```
final_xgb = xgb.XGBRegressor(
    objective="reg:squarederror",
    eval_metric="rmse",
    n_estimators=5000,
    learning_rate=0.03,
    max_depth=3,
    min_child_weight=6,
    subsample=0.7,
    colsample_bytree=0.7,
    reg_alpha=0.1,
    reg_lambda=3.0,
    gamma=0.1,
    random_state=42,
    n_jobs=-1,
    tree_method="hist"
)
```

Train on FULL training data (no validation now)

model choice and complexity were already validated using CV now we want maximum data usage for best prediction

In [36]:
```
final_xgb.fit(X_full, y_full_log, verbose=False)
```

Out[36]:

▾ XGBRegressor

▶ Parameters

Predict on test set (log-scale → original prices)

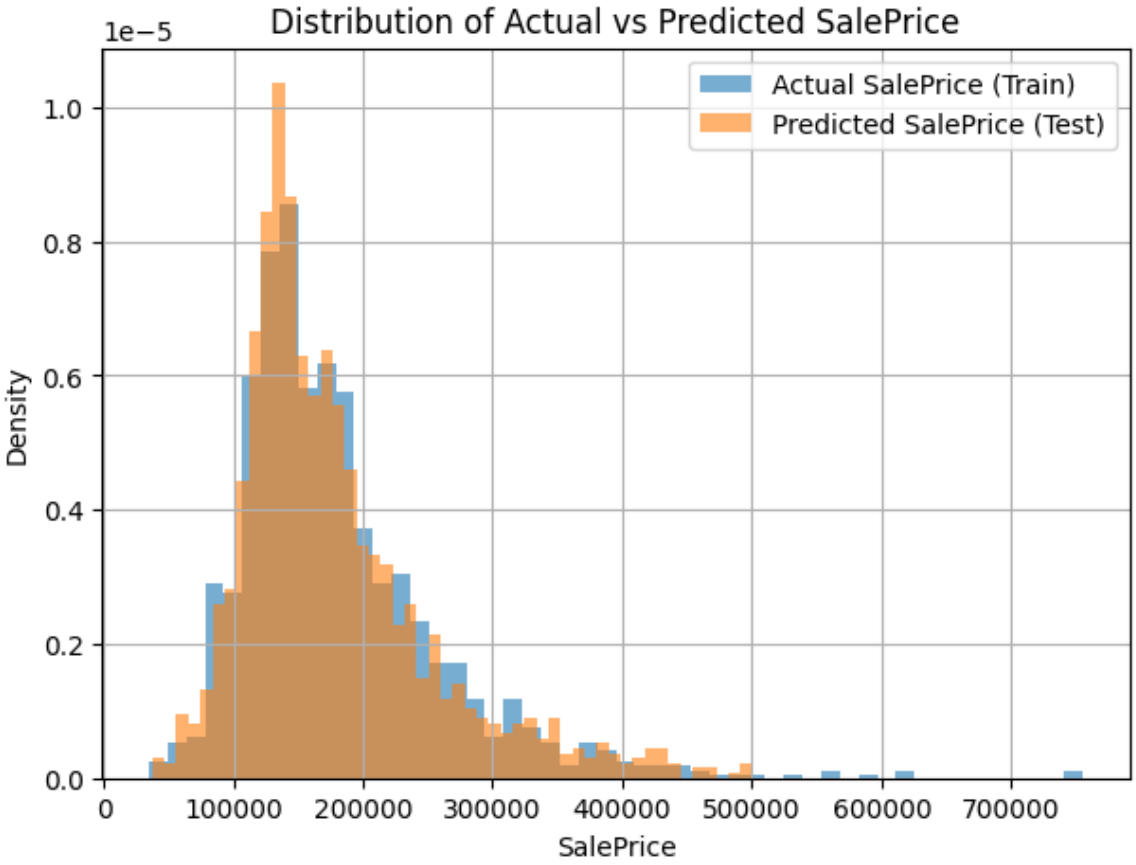
In [37]:
```
# Predict in log space
test_pred_log = final_xgb.predict(X_test_final)

# Convert back to original SalePrice scale
test_pred = np.expm1(test_pred_log)
```

```
print("Prediction summary:")
print(pd.Series(test_pred).describe())
```

```
Prediction summary:
count      1459.000000
mean     177664.531250
std       75251.187500
min       37627.167969
25%      128403.644531
50%      158303.812500
75%      206767.796875
max      500519.843750
dtype: float64
```

In [ ]:
```
# True SalePrice from training data (original scale)
y_train_actual = np.expm1(y_full_log)

# Predicted SalePrice from test set
y_test_pred = test_pred
```

In [39]:
```
plt.figure(figsize=(7,5))

plt.hist(
    y_train_actual,
    bins=50,
    alpha=0.6,
    density=True,
    label="Actual SalePrice (Train)"
)

plt.hist(
    y_test_pred,
    bins=50,
    alpha=0.6,
    density=True,
    label="Predicted SalePrice (Test)"
)

plt.xlabel("SalePrice")
plt.ylabel("Density")
plt.title("Distribution of Actual vs Predicted SalePrice")
plt.legend()
plt.grid(True)

plt.show()
```



In [40]:
```
print("Actual SalePrice (train):")
print(pd.Series(y_train_actual).describe())

print("\nPredicted SalePrice (test):")
print(pd.Series(y_test_pred).describe())
```

```
Actual SalePrice (train):
count      1460.000000
mean     180921.195890
std       79442.502883
min       34900.000000
25%      129975.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64

Predicted SalePrice (test):
count      1459.000000
mean     177664.531250
std       75251.187500
min       37627.167969
25%      128403.644531
50%      158303.812500
75%      206767.796875
max      500519.843750
dtype: float64
```

Explanation of the distribution plot

The figure compares the distribution of actual SalePrice values from the

training data with the predicted SalePrice values on the test data.

1️⃣ Overall shape

Both distributions are right-skewed, meaning: Most houses are priced in the lower–middle range Fewer houses appear at very high prices This is expected for housing prices and shows that the model learned the overall structure of the data correctly.

2️⃣ Central tendency (mean & median)

Statistic	Actual (Train)	Predicted (Test)
Mean	180,921	177,665
Median	163,000	158,304

The predicted mean and median are very close to the actual values. This indicates that the model is not systematically overestimating or underestimating house prices.

3️⃣ Spread of values (standard deviation & quartiles)

Statistic	Actual	Predicted
Std	79,443	75,251
25%	129,975	128,404
75%	214,000	206,768

The predicted distribution is slightly narrower, which is normal and desirable:

- Extreme prices are harder to predict
- Regularized models tend to be more conservative
- This behavior helps reduce overfitting

4️⃣ High-price tail behavior (important)

Actual prices go up to ~755,000 Predicted prices go up to ~500,000

This shows that the model avoids extreme predictions. This is a sign of controlled overfitting, not a mistake.

In real applications, it is usually better to:

slightly under-predict extreme values than to overfit noise.

Final interpretation

The predicted SalePrice distribution closely matches the real distribution in

shape, center, and range, while being slightly smoother and more conservative. This indicates that the model learned meaningful price patterns and generalizes well without producing unrealistic predictions.

Creating submission file as for final prediction

In [41]:
```
test_df = pd.read_csv("data/test.csv")
test_ids = test_df["Id"]
```

In [42]:
```
submission = pd.DataFrame({
    "Id": test_ids,
    "SalePrice": test_pred
})

submission.to_csv("TestSetPredictions_xgboost.csv", index=False)
print("TestSetPredictions_xgboost.csv saved successfully!")
```

```
TestSetPredictions_xgboost.csv saved successfully!
```

Final Conclusion

In this project, I built and evaluated several regression models to predict apartment sale prices using the Ames Housing dataset. The process included data cleaning, feature engineering, one-hot encoding, and applying a log transformation to the target variable to handle skewness. Since the dataset is not very large and contains many features, a major focus of the project was identifying and reducing overfitting rather than simply maximizing prediction accuracy.

To address this, three strong five-based models—XGBoost, LightGBM, CatBoost, Random Forest, Ensembling-were compared using 5-fold cross-validation and log-RMSE as the evaluation metric. Instead of looking only at validation error, I also examined the gap between training and validation performance to better understand how well each model generalizes. CatBoost achieved the lowest validation error, but it showed a large gap between training and validation scores, indicating that it was overfitting the training data. LightGBM performed slightly better than XGBoost in terms of validation error, but it still showed noticeable overfitting.

XGBoost provided the most balanced results. Although its validation error was not the lowest, it consistently showed the smallest and most stable difference between training and validation errors across folds. This indicated a better balance between bias and variance and more reliable generalization to unseen data. For this reason, XGBoost was selected as the final model.

The final XGBoost model was trained on the full training dataset and used to generate predictions for the test set. A comparison of the predicted and

actual price distributions showed similar shapes and ranges, with predictions being slightly more conservative for extreme values. This behavior suggests that the model learned meaningful patterns in the data while avoiding excessive memorization.

Overall, this project helped me better understand the full machine learning workflow and, more importantly, the trade-offs between model complexity, performance, and overfitting. Rather than choosing a model based only on the lowest error, this work emphasizes the importance of selecting models that generalize well, especially when working with limited data.