- have not tried replacing it with its approximations, such as LinearSVC and SGDClassifier

In [5]:
```
baseline_models = {
    'Logistic Regression': LogisticRegression(random_state=42, max_iter=1000
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=4
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100, random
    #'SVM': SVC(probability=True, random_state=42),
    'Neural Network': MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=1
    'XGBoost': XGBClassifier(n_estimators=100, random_state=42, use_label_er
    'LightGBM': LGBMClassifier(n_estimators=100, random_state=42, verbosity=
}


baseline_results = {}

for name, model in baseline_models.items():
    print(f"\nTraining {name}...")
    start_time = time.time()

    # Train model
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)
    y_pred_proba = model.predict_proba(X_test)[:, 1]

    # Calculate metrics
    baseline_results[name] = {
        'model': model,
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred),
        'recall': recall_score(y_test, y_pred),
        'f1': f1_score(y_test, y_pred),
        'roc_auc': roc_auc_score(y_test, y_pred_proba),
        'avg_precision': average_precision_score(y_test, y_pred_proba),
        'train_time': time.time() - start_time
    }

    print(f"ROC-AUC: {baseline_results[name]['roc_auc']:.4f}")
    print(f"F1 Score: {baseline_results[name]['f1']:.4f}")
    print(f"Training time: {baseline_results[name]['train_time']:.2f} seconc

# Display baseline results
baseline_df = pd.DataFrame(baseline_results).T
print("\nBaseline Model Comparison:")
print(baseline_df.round(4))
```

```
Training Logistic Regression...
ROC-AUC: 0.7647
F1 Score: 0.5517
Training time: 1.23 seconds

Training Decision Tree...
ROC-AUC: 0.7283
F1 Score: 0.5832
Training time: 1.03 seconds

Training Random Forest...
ROC-AUC: 0.7640
F1 Score: 0.5923
Training time: 4.33 seconds

Training Gradient Boosting...
ROC-AUC: 0.7606
F1 Score: 0.6040
Training time: 17.47 seconds

Training Neural Network...
ROC-AUC: 0.7565
F1 Score: 0.6167
Training time: 33.45 seconds

Training XGBoost...
ROC-AUC: 0.7652
F1 Score: 0.6030
Training time: 1.25 seconds

Training LightGBM...
ROC-AUC: 0.7653
F1 Score: 0.6017
Training time: 1.70 seconds

Baseline Model Comparison:
                                                              model  \
Logistic Regression  LogisticRegression(max_iter=1000, random_state...
Decision Tree                  DecisionTreeClassifier(random_state=42)
Random Forest        (DecisionTreeClassifier(max_features='sqrt', r...
Gradient Boosting    ([DecisionTreeRegressor(criterion='friedman_ms...
Neural Network       MLPClassifier(hidden_layer_sizes=(100, 50), ma...
XGBoost              XGBClassifier(base_score=None, booster=None, c...
LightGBM                 LGBMClassifier(random_state=42, verbosity=-1)

                     accuracy precision    recall        f1   roc_auc  \
Logistic Regression  0.990569  0.889974  0.399708  0.551655  0.764716
Decision Tree        0.990904  0.871005  0.438304  0.583155  0.728328
Random Forest        0.990976  0.860647  0.451462  0.592252  0.763961
Gradient Boosting    0.991108  0.854089  0.467251  0.604045  0.760632
Neural Network       0.991189  0.836673  0.488304  0.616691    0.7565
XGBoost              0.991138  0.861957  0.463743  0.603042  0.765199
LightGBM             0.991087  0.856371  0.463743  0.601669  0.765316

                    avg_precision train_time
Logistic Regression      0.489116   1.228961
```

```
Decision Tree            0.445948   1.028317
Random Forest            0.507223   4.334978
Gradient Boosting        0.505698  17.471276
Neural Network           0.507831   33.44746
XGBoost                  0.512217   1.253606
LightGBM                 0.512047   1.696867
```

In [6]:
```
baseline_df.to_csv('Baseline Model Performance.csv')
```

For Fraud Detection, Recall (aka True Positive Rate or Sensitivity) is crucial that we want to catch as many fraud cases as possible, i.e.the cost of missing a true positive is high. However, it can also increase the risk of false alarms. Thus, to choose the proper models for further training, we have to consider all metrics and prioritize certain metrics over others. After a glance of the above results, GBDT and NN are the two choices for me, just like what we have done in the paper years ago. For future efficiency, let's utilize self-defined function combining multiple metrics with customizable weights.

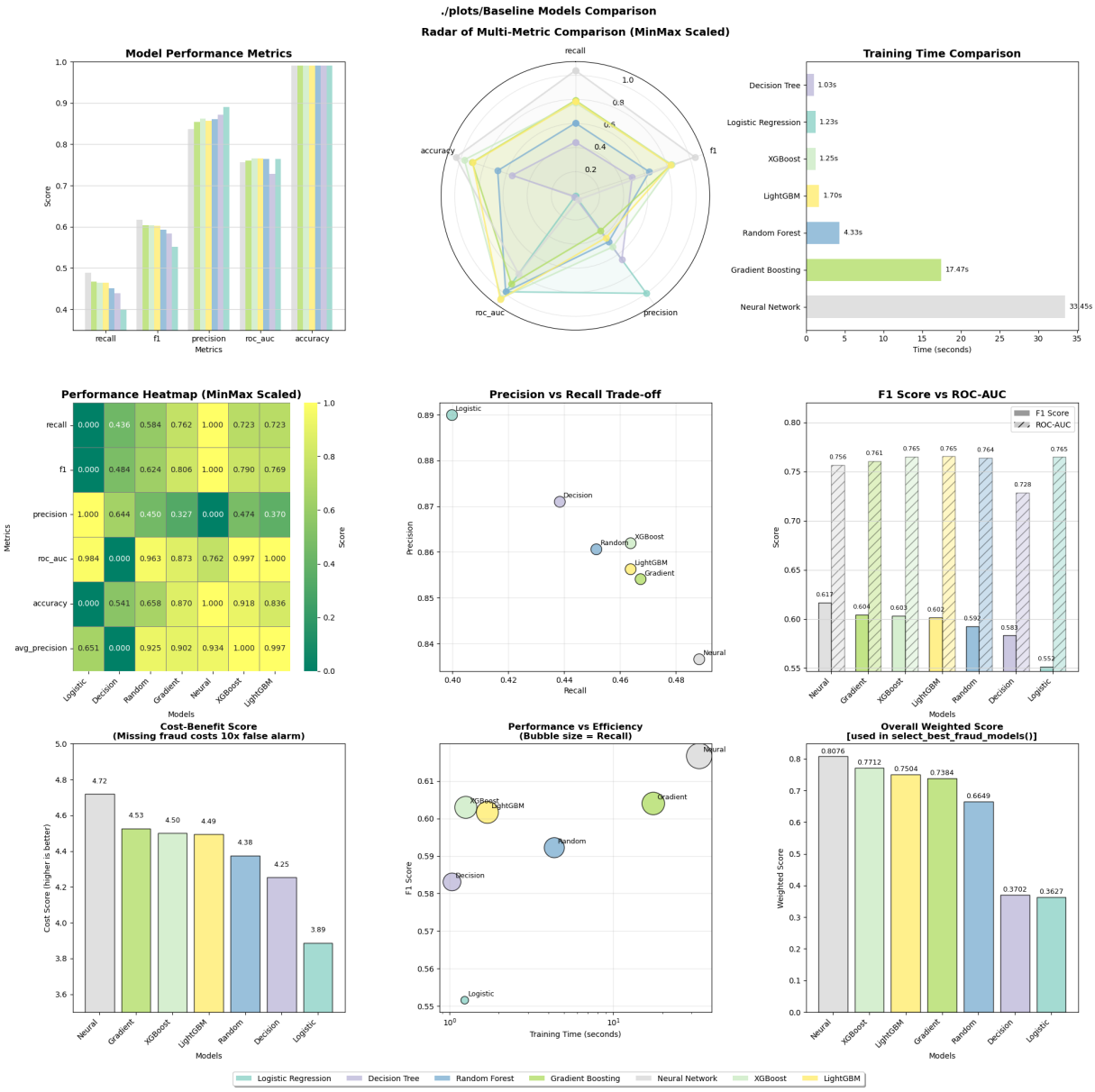
- and training time may need to be taken into consideration in real situation

$$Recall = \frac{TP}{TP + FN}$$$$Precision = \frac{TP}{TP+FP}$$$$F1 = 2 \times \frac{Recall \times Precision}{Recall+Precision}$$

In [5]:
```
baseline_df = pd.read_csv('Baseline Model Performance.csv', index_col=0)
#baseline_df
```

In [6]:
```
fraud_weights = {
        'recall': 0.35,       # Most important - catch fraud cases
        'f1': 0.25,           # Balance of precision and recall
        'precision': 0.15,    # Avoid too many false alarms
        'roc_auc': 0.15,      # Overall discrimination ability
        'avg_precision': 0.10 # Performance across thresholds
    }
```

In [7]:
```
from func import plot_model_comparison
plot_model_comparison(baseline_df, weights = fraud_weights, top_n=3,
                      figsize=(20, 20), figsave=True, figtitle = './plots/Ba
```



In [8]:
```
from func import select_best_fraud_models
top_models, weighted_score = select_best_fraud_models(baseline_df, top_n=3,
```

In [9]:
```
top_models
```

Out[9]:
```
['Neural Network', 'XGBoost', 'LightGBM']
```

In [10]:
```
weighted_score
```

Out[10]:
```
Logistic Regression    0.362708
Decision Tree          0.370180
Random Forest          0.664941
Gradient Boosting      0.738394
Neural Network         0.807627
XGBoost                0.771177
LightGBM               0.750403
Name: weighted_score, dtype: float64
```

Hyperparameter Optimization

the top 3 models for optimization are selected based on baseline performance, and I also want to add GBDT

In [11]:
```
top_models = top_models + ['Gradient Boosting']
```

skip NN optimization due to large dataset and much time, and for fraud detection, tree models are better anyway

In [12]:
```
top_models.remove('Neural Network')
top_models
```

Out[12]:
```
['XGBoost', 'LightGBM', 'Gradient Boosting']
```

> **a little plug here**, to train faster, if there is cuda GPUS (usually windows OS), can use:
>
> - XGBClassifier(tree_method='gpu_hist', ...)
> - LGBMClassifier(device='gpu', ...)
> - NN: switch to PyTorch or TensorFlow
> - SVM: RAPIDS cuML or ThunderSVM
>
> from the above, for mac silicon, except NN (tf is recommended than pytorch), others are not supported

optimize_all_models_parallel(..., method='hyperopt' or 'optuna', 'randomized', n_trials=100,...)

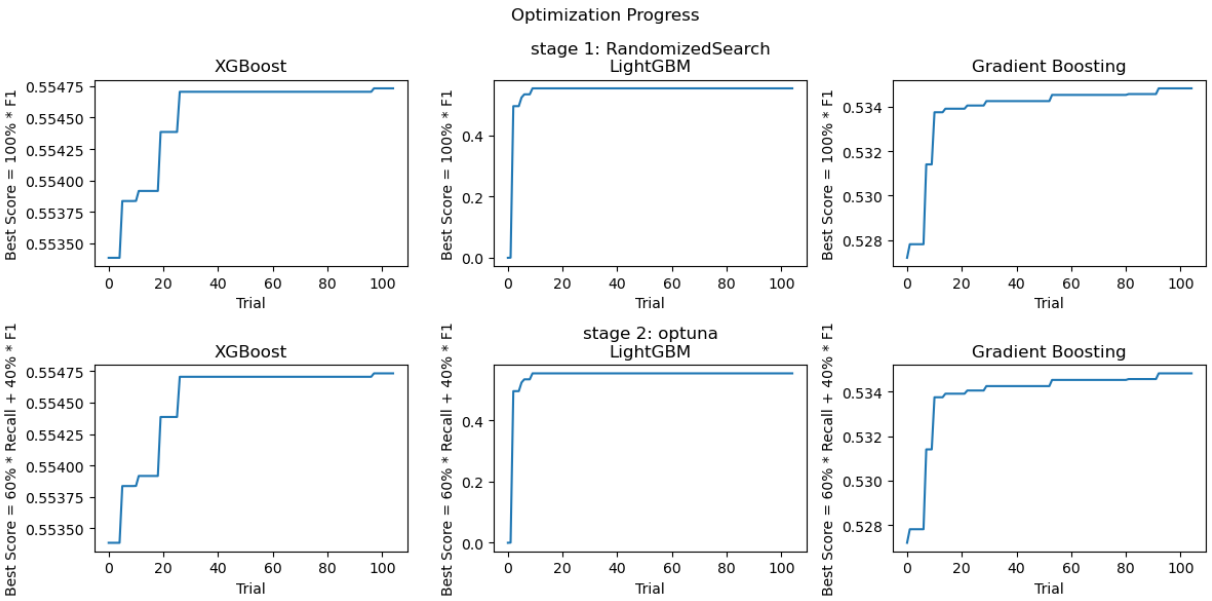
In [15]:
```
from func_2stageOpt import two_stage_optimize_multiple
optimized_models = two_stage_optimize_multiple(
    X_train, y_train,
    models = top_models,
    n_trials = 150,   # 150 trials per model
    stage1_ratio=0.3,  # 30% for exploration
    show_trial_details=False  # Clean output
)
```

```
Running Stage 1...
Stage 1 complete! (Time: 9718.3s)
Best F1 Score: 0.6140

========================================
Stage 2: Focused search around best parameters
========================================

Running Stage 2 optimization...
  Progress: 1% (1/105 trials) - Best: 0.5272 - Time: 141.2s
  Progress: 10% (11/105 trials) - Best: 0.5337 - Time: 7634.2s
  Progress: 20% (21/105 trials) - Best: 0.5339 - Time: 10101.2s
  Progress: 30% (31/105 trials) - Best: 0.5342 - Time: 15769.1s
  Progress: 39% (41/105 trials) - Best: 0.5342 - Time: 18088.6s
  Progress: 49% (51/105 trials) - Best: 0.5342 - Time: 20640.5s
  Progress: 58% (61/105 trials) - Best: 0.5345 - Time: 22844.9s
  Progress: 68% (71/105 trials) - Best: 0.5345 - Time: 25605.5s
  Progress: 77% (81/105 trials) - Best: 0.5345 - Time: 30073.5s
  Progress: 87% (91/105 trials) - Best: 0.5346 - Time: 37253.3s
  Progress: 96% (101/105 trials) - Best: 0.5348 - Time: 40416.2s

============================================================
Optimization Complete!
============================================================
Stage 1: 0.6140 (Time: 9718.3s)
Stage 2: 0.5348 (Time: 41345.9s)
Improvement: -0.0792
Total time: 51064.2s

================================================================================
====
OPTIMIZATION SUMMARY
================================================================================
====
Model                Stage 1    Stage 2    Improvement  Time (s)
--------------------------------------------------------------------------------
----
XGBoost              0.6099     0.5547     -0.0552      3391.5
LightGBM             0.6150     0.5530     -0.0620      12640.6
Gradient Boosting    0.6140     0.5348     -0.0792      51064.2
```

In [ ]:
```python
# and less trials for NN
from func_2stageOpt import two_stage_optimization
optimized_NN = two_stage_optimization(
    X_train, y_train,
    model_name='Neural Network',
    stage1_trials=15,  # for exploration
    stage2_trials=30,  # for exploitation
    range_factor=0.5,  # Search ±50% around best values
    verbose=True,
    show_trial_details=False  # Clean output
)
```

In [19]:
```python
optimized_models
```

In [42]:
```
fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(12, 6))
i = 0
for name, model in optimized_models.items():
    ax[0, i].plot(model['stage2_history'])
    ax[0, i].set_xlabel('Trial')
    ax[0, i].set_ylabel('Best Score = 100% * F1')
    if i == 1:
        title = f'stage 1: RandomizedSearch\n{name}'
    else:
        title = f'\n{name}'
    ax[0, i].set_title(title)

    ax[1, i].plot(model['stage2_history'])
    ax[1, i].set_xlabel('Trial')
    ax[1, i].set_ylabel('Best Score = 60% * Recall + 40% * F1')
    if i == 1:
        title = f'stage 2: optuna\n{name}'
    else:
        title = f'\n{name}'
    ax[1, i].set_title(title)
    i += 1
plt.suptitle('Optimization Progress')
plt.tight_layout()
plt.show()
```



Calculate the Optimized Models' Performance with the best parameters

load dataset again if the previous has been deleted

in the result dataset, the models are stored as string in the model row, need to convert them to real model objects to be trained

In [8]:
```
# optimized_models = pd.read_csv('Top3 Optimized Results.csv', index_col=0)
# optimized_models
# Convert string to actual model objects
# optimized_models = {name: eval(model_str) for name, model_str in dict(opti
```

In [25]:
```
optimized_results = {}

for name, model in optimized_models.items():
    model = model['model']
    print(f"\nTraining {name}...")
    start_time = time.time()

    # Train model
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)
    y_pred_proba = model.predict_proba(X_test)[:, 1]

    # Calculate metrics
    optimized_results[name] = {
        'model': model,
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred),
        'recall': recall_score(y_test, y_pred),
        'f1': f1_score(y_test, y_pred),
        'roc_auc': roc_auc_score(y_test, y_pred_proba),
        'avg_precision': average_precision_score(y_test, y_pred_proba),
        'train_time': time.time() - start_time
    }

    print(f"ROC-AUC: {optimized_results[name]['roc_auc']:.4f}")
    print(f"F1 Score: {optimized_results[name]['f1']:.4f}")
    print(f"Training time: {optimized_results[name]['train_time']:.2f} secor

# Display optimized models results
optimized_df = pd.DataFrame(optimized_results).T
print("\nOptimized Model Comparison:")
print(optimized_df.round(4))
```

```
Training XGBoost...
ROC-AUC: 0.7708
F1 Score: 0.6181
Training time: 5.33 seconds

Training LightGBM...
ROC-AUC: 0.7702
F1 Score: 0.6166
Training time: 12.13 seconds

Training Gradient Boosting...
ROC-AUC: 0.7702
F1 Score: 0.6191
Training time: 64.94 seconds

Optimized Model Comparison:
                                                          model  \
XGBoost            XGBClassifier(base_score=None, booster=None, c...
LightGBM           LGBMClassifier(bagging_fraction=0.357144695323...
Gradient Boosting  ([DecisionTreeRegressor(criterion='friedman_ms...

                   accuracy precision    recall        f1   roc_auc  \
XGBoost            0.990671  0.751267  0.525089  0.618138  0.770787
LightGBM           0.990637  0.749789  0.523613  0.616615  0.770164
Gradient Boosting  0.991397  0.852043  0.486128  0.619057  0.770245

                  avg_precision train_time
XGBoost                0.522613    5.32891
LightGBM               0.479162   12.12569
Gradient Boosting       0.52467  64.935538
```

In [26]:
```
optimized_df.to_csv('Optimized Model Performance.csv')
```

Baseline vs Optimized

check how much they have been optimized from their corresponding baseline models, to make sure the optimization is meaningful. Otherwise, run the optimization again with larger trials or try other method, like hyperopt

In [43]:
```
baseline_df.loc[optimized_df.index, :]
```

Out[43]:

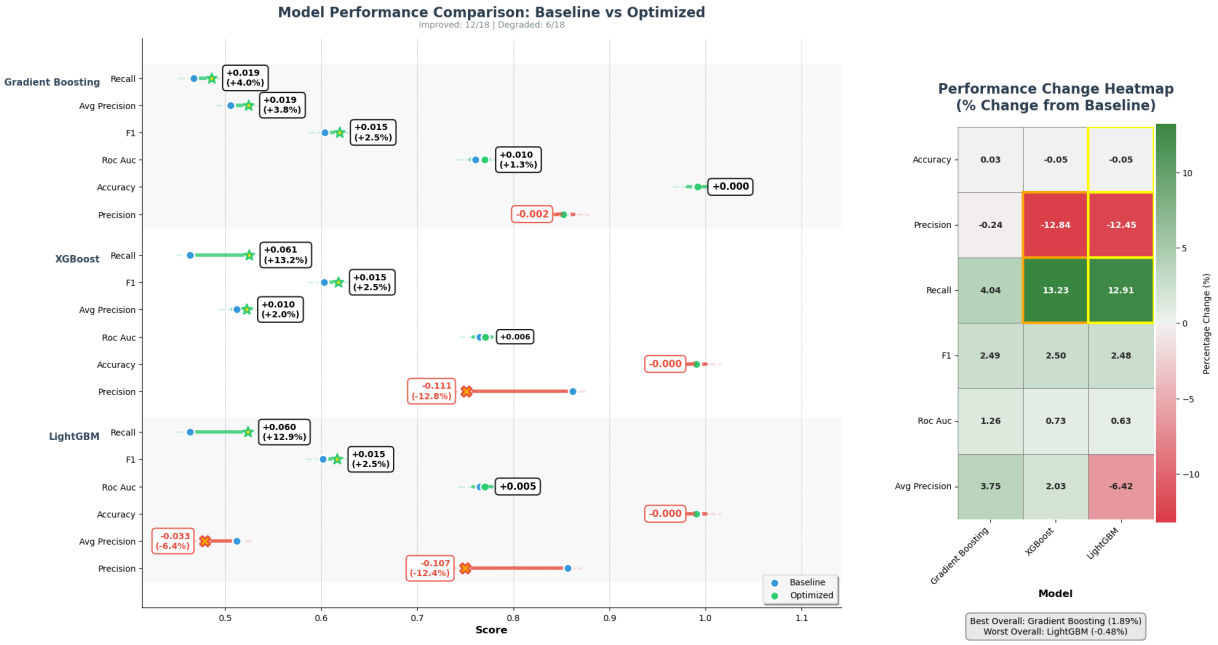
	model	accuracy	precision	recall	
XGBoost	XGBClassifier(base_score=None, booster=None, c...	0.991138	0.861957	0.463743	C
LightGBM	LGBMClassifier(random_state=42, verbosity=-1)	0.991087	0.856371	0.463743	C
Gradient Boosting	GradientBoostingClassifier(random_state=42)	0.991108	0.854089	0.467251	C

In [44]:
```
optimized_df
```

Out[44]:

	model	accuracy	precision	re
XGBoost	XGBClassifier(base_score=None, booster=None, c...	0.990671	0.751267	0.525
LightGBM	LGBMClassifier(bagging_fraction=0.357144695323...	0.990637	0.749789	0.523
Gradient Boosting	([DecisionTreeRegressor(criterion='friedman_ms...	0.991397	0.852043	0.486

In [45]:
```
from func import compare_performance
baseline_df = pd.read_csv('Baseline Model Performance.csv', index_col=0)
#optimized_df = pd.read_csv('Optimized Model Performance.csv', index_col=0)
compare_performance(baseline_df, optimized_df
                    , save_path = './plots/Optimization Effect.png'
                   )
```




```
Model Performance Summary:
              Model  Metrics Improved  Metrics Degraded Avg Change Best Improv
ement         Worst Change
Gradient Boosting                  5                 1      1.89%  recall (+
4.04%)  precision (-0.24%)
          XGBoost                  4                 2      0.93% recall (+1
3.23%) precision (-12.84%)
         LightGBM                  3                 3     -0.48% recall (+1
2.91%) precision (-12.45%)
```

even the precision decreased for LightGBM and XGB, the recall and a lot scores of them all improved, so just move on to next step

Compare Optimized Models

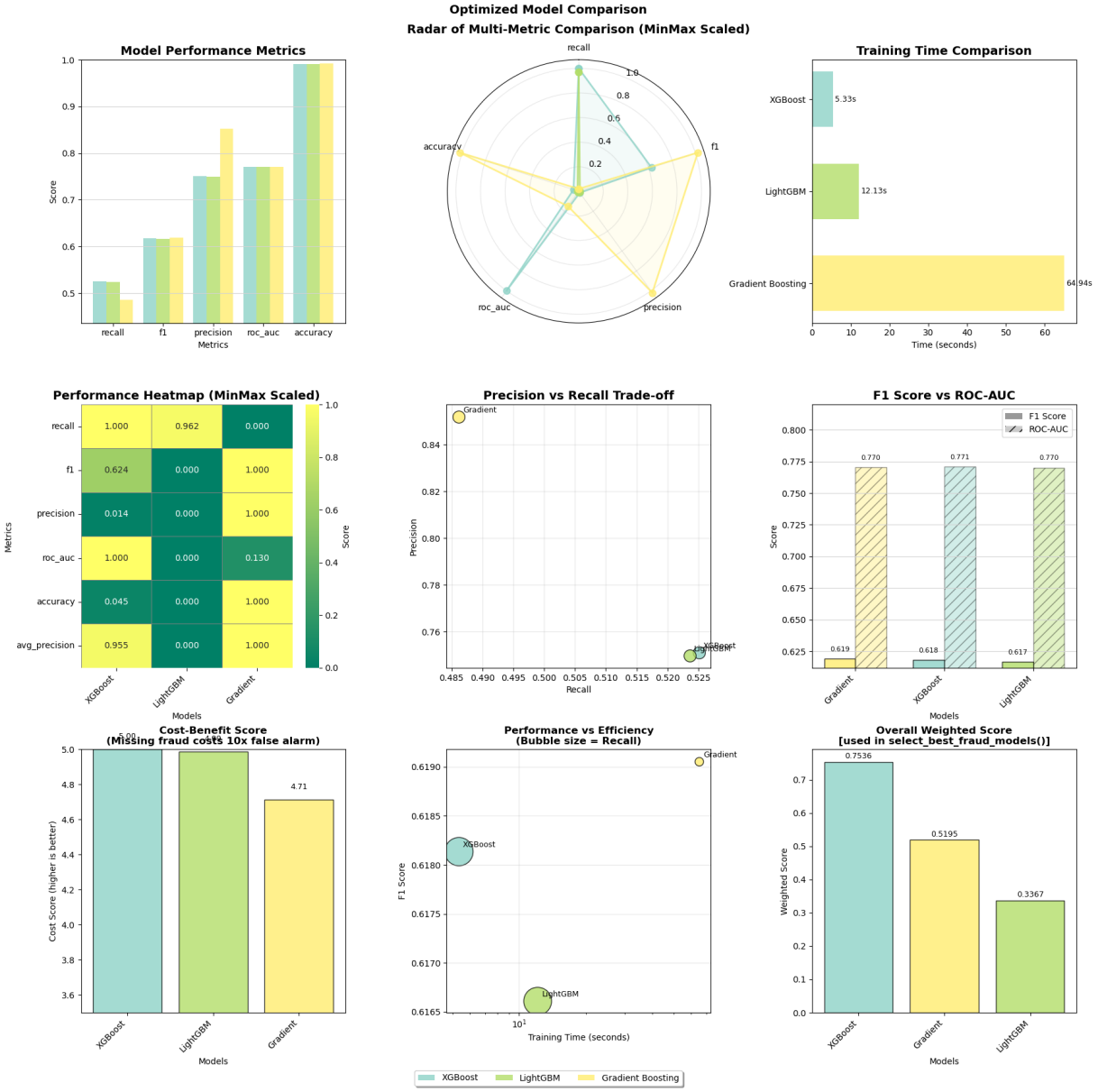
again, auto-select the best model, only one for this time

In [46]:
```
#optimized_df = pd.read_csv('Optimized Model Performance.csv', index_col=0)
```

In [47]:
```
fraud_weights = {
        'recall': 0.35,       # Most important - catch fraud cases
        'f1': 0.25,           # Balance of precision and recall
        'precision': 0.15,    # Avoid too many false alarms
        'roc_auc': 0.15,      # Overall discrimination ability
        'avg_precision': 0.10 # Performance across thresholds
    }
```

In [48]:
```
from func import select_best_fraud_models
best_model_name, weighted_score = select_best_fraud_models(optimized_df, top
```

In [49]:
```
from func import plot_model_comparison
plot_model_comparison(optimized_df, weights = fraud_weights, top_n=1,
                      figsize=(18, 18), figsave=True, figtitle = 'Optimized
```



Overall performance suggest to choose XGBoost

Final Evaluation

In [53]:
```
# optimized_models = pd.read_csv('Top3 Optimized Results.csv', index_col=0)
# Convert string to actual model objects if read from csv
# optimized_models = {name: eval(model_str) for name, model_str in dict(opti
best_model = optimized_models[best_model_name[0]]['model']
best_model
```

Out[53]:

XGBClassifier

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=0.6238481155990337, device=None,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric='logloss', feature_types=None,
              gamma=1.5531831263616995, grow_policy=None, importanc
e_type=None,
              interaction_constraints=None, learning_rate=0.0030976
135766664927,
```

In [51]:
```
z_scaled = pd.read_csv('./data/z_scaled.csv', index_col=0)
z_scaled = optimize_dtypes(z_scaled)
```

```
OPTIMIZING DATA TYPES
----------------------------------------
Starting memory usage: 299.59 MB
Ending memory usage: 150.73 MB
Memory reduction: 49.7%
```

In [54]:
```
# Evaluate model
from func import evaluate_model_goodness
goodness_summary, detailed_metrics = evaluate_model_goodness(
    model=best_model, model_name=best_model_name[0],
    whole_dataset = z_scaled, test_size=0.2,
    top_percent=0.03, niter_max=5,
    display_confusion_matrices=False
)
```

```
Evaluating XGBoost performance over 5 iterations...


XGBoost Goodness Summary (Average over 5 iterations):
```

	train	test	oot
FDR	0.542	0.5352	0.5346
KS	0.5228	0.5174	0.5159
AUC	0.7688	0.7643	0.7611
Thresholds	0.2568	0.2924	0.326
Accuracy	0.9905	0.9905	0.9901
Misclassification	0.0095	0.0095	0.0099
False Positive Rate	0.0026	0.0026	0.0028
True Positive Rate	0.5203	0.5136	0.5143
True Negative Rate	0.9974	0.9974	0.9972
Precision	0.7493	0.7461	0.7352

Key Metrics Stability (Standard Deviation):

	train_std	test_std	oot_std
AUC	0.0005	0.0031	0.0009
KS	0.001	0.0043	0.0002
FDR	0.0013	0.0051	0.0
ACC	0.0001	0.0001	0.0
PRE	0.0049	0.0081	0.0033

Compare with the GBDT model buit years ago:

```
display(GBDT_goodness)
```

	train	test	oot
FDR	0.5686	0.6079	0.5436
KS	0.5489	0.5357	0.5233
AUC	0.7948	0.7832	0.7762
Thresholds	0.0137	0.0129	0.0116
Accuracy	0.9925	0.9912	0.9907
Misclassification	0.0075	0.0088	0.0093
False Positive Rate	0.0005	0.0012	0.0014
True Positive Rate	0.5139	0.4711	0.4512
True Negative Rate	0.9995	0.9988	0.9986
Precision	0.9399	0.8546	0.8208

1. Discrimination Ability (AUC)

- XGBoost shows lower AUC across all sets (train: 0.7688, test: 0.7643, oot: 0.7611)
- GBDT had higher AUC (train: 0.7948, test: 0.7832, oot: 0.7762)

This suggests GBDT has better overall discrimination between fraud and non-fraud cases

2. False Positive Rate - Critical for Fraud Detection

- XGBoost: Much higher FPR (0.0026-0.0028)
- GBDT: Significantly lower FPR (0.0005-0.0014)

This is concerning - XGBoost flags 2-5x more legitimate transactions as fraudulent

3. Precision Trade-off

- XGBoost: Lower precision (0.7352-0.7493)
- GBDT: Higher precision (0.8208-0.9399)

GBDT catches proportionally more actual fraud among flagged transactions

4. Model Stability

- XGBoost shows smaller train-test gaps, suggesting less overfitting
- GBDT shows larger performance drops from train to test/oot sets

Overall Assessment
For fraud detection, GBDT appears superior because:

- Lower false positives - Critical for customer experience (fewer legitimate transactions blocked)
- Higher precision - When it flags fraud, it's more likely to be correct

Better AUC - Overall better at ranking fraud probability

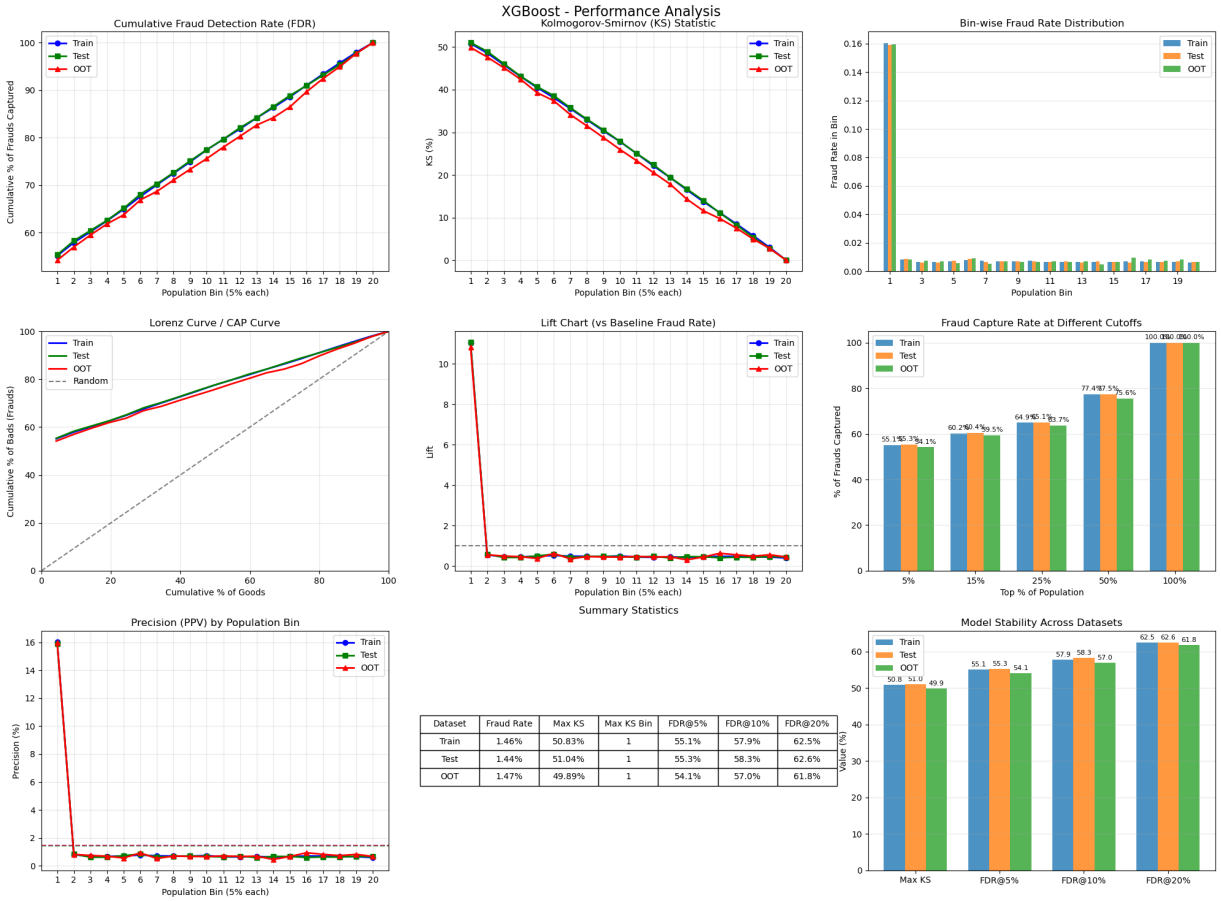
The XGBoost model's higher false positive rate is particularly problematic in fraud detection, where blocking legitimate transactions can damage customer trust and business relationships.

Recommendation:
Unless there are compelling operational reasons (like inference speed or model size), I would recommend sticking with the GBDT model for this fraud detection use case. The significantly lower false positive rate and higher precision make it more suitable for production deployment.

In [55]:
```
from func import calculate_performance_forms, plot_performance_metrics
forms_dict = calculate_performance_forms(
    model=best_model,
    whole_dataset=z_scaled,
    n_bins=20,
    model_name=best_model_name[0]
)
```

```
XGBoost – Data Summary:
Train: 541,747 goods, 8,001 bads, fraud rate: 1.46%
Test:  232,219 goods, 3,388 bads, fraud rate: 1.44%
OOT:   193,445 goods, 2,894 bads, fraud rate: 1.47%
```

In [56]:
```
plot_performance_metrics(forms_dict, model_name=best_model_name[0])
```



Summary Statistics

Dataset	Fraud Rate	Max KS	Max KS Bin	FDR@5%	FDR@10%	FDR@20%
Train	1.46%	50.83%	1	55.1%	57.9%	62.5%
Test	1.44%	51.04%	1	55.3%	58.3%	62.6%
OOT	1.47%	49.89%	1	54.1%	57.0%	61.8%

XGBoost – Train Form (Top 5 bins):

	pop_bin	bin_records	bin_goods	bin_bads	bin_%good	bin_%bad	cum_records	c
0	1	27488	23080	4408	0.8396	0.1604	27488	
1	2	27488	27266	222	0.9919	0.0081	54976	
2	3	27488	27305	183	0.9933	0.0067	82464	
3	4	27488	27303	185	0.9933	0.0067	109952	
4	5	27488	27293	195	0.9929	0.0071	137440	

XGBoost – Test Form (Top 5 bins):

	pop_bin	bin_records	bin_goods	bin_bads	bin_%good	bin_%bad	cum_records	c
0	1	11781	9907	1874	0.8409	0.1591	11781	
1	2	11781	11681	100	0.9915	0.0085	23562	
2	3	11781	11708	73	0.9938	0.0062	35343	
3	4	11781	11708	73	0.9938	0.0062	47124	
4	5	11781	11695	86	0.9927	0.0073	58905	

XGBoost – OOT Form (Top 5 bins):

	pop_bin	bin_records	bin_goods	bin_bads	bin_%good	bin_%bad	cum_records	c
0	1	9817	8250	1567	0.8404	0.1596	9817	
1	2	9817	9736	81	0.9917	0.0083	19634	
2	3	9817	9744	73	0.9926	0.0074	29451	
3	4	9817	9749	68	0.9931	0.0069	39268	
4	5	9817	9762	55	0.9944	0.0056	49085	

In [57]:
```
# train_form = forms_dict['train_form']
# test_form = forms_dict['test_form']
oot_form = forms_dict['oot_form']
oot_form
```

Out[57]:

	pop_bin	bin_records	bin_goods	bin_bads	bin_%good	bin_%bad	cum_records
0	1	9817	8250	1567	0.8404	0.1596	9817
1	2	9817	9736	81	0.9917	0.0083	19634
2	3	9817	9744	73	0.9926	0.0074	29451
3	4	9817	9749	68	0.9931	0.0069	39268
4	5	9817	9762	55	0.9944	0.0056	49085
5	6	9817	9726	91	0.9907	0.0093	58902
6	7	9817	9765	52	0.9947	0.0053	68719
7	8	9817	9748	69	0.9930	0.0070	78536
8	9	9817	9751	66	0.9933	0.0067	88353
9	10	9817	9752	65	0.9934	0.0066	98170
10	11	9817	9747	70	0.9929	0.0071	107987
11	12	9817	9751	66	0.9933	0.0067	117804
12	13	9817	9749	68	0.9931	0.0069	127621
13	14	9817	9772	45	0.9954	0.0046	137438
14	15	9817	9751	66	0.9933	0.0067	147255
15	16	9817	9725	92	0.9906	0.0094	157072
16	17	9817	9736	81	0.9917	0.0083	166889
17	18	9817	9745	72	0.9927	0.0073	176706
18	19	9817	9736	81	0.9917	0.0083	186523
19	20	9816	9750	66	0.9933	0.0067	196339

In [ ]: