Step 1: Load Required Libraries

I begin by importing essential Python libraries that provide the building blocks for our analysis. `networkx` is used to create and manipulate graphs, offering a wide suite of graph algorithms and utilities. `numpy` and `pandas` are used for numerical operations and tabular data handling. To support visual analytics, I use `plotly` to generate interactive network plots, enabling better interpretability. Additionally, `datetime` and `random` are leveraged to simulate transaction attributes like timestamps and amounts, while `scikit-learn` helps with scaling and other numerical preprocessing tasks. These tools collectively support the end-to-end workflow for detecting patterns in networked data.

In [1]:
```
import datetime
import numpy as np
import pandas as pd
import networkx as nx
import plotly.graph_objects as go
from sklearn.preprocessing import MinMaxScaler
from networkx.algorithms.community import label_propagation_communities
```

Step 2: Generate Base Transaction Network

In this step, I construct a synthetic transaction network using the Barabási–Albert (BA) model. This model is widely used in network science to generate graphs with scale-free properties. Scale-free networks resemble real-world transaction and communication networks because they contain a few high-degree 'hub' nodes, while the majority of nodes have relatively few connections. These structural features make the network more realistic and help us evaluate how fraud detection algorithms perform in practical scenarios. The nodes in the graph represent entities such as accounts or users, and the edges represent financial transactions or other interactions.

In [2]:
```
def generate_transaction_graph_modified(n=300, m=2):
    return nx.barabasi_albert_graph(n=n, m=m, seed=42)

undirected_graph = generate_transaction_graph_modified()
```

Step 3: Inject Fraud Ring

To simulate fraudulent behavior within the network, I injected a synthetic fraud ring, which is a group of nodes that are all connected to one another. This setup mimics collusive behavior where a group of actors coordinate their actions to evade detection. In a real-world setting, this might represent a set of accounts under common control or users involved in a money-laundering operation. By creating a dense cluster of interconnections, I introduced a known anomaly into the network, which we can later attempt to detect using graph analytics.

In [3]:
```
def inject_fraud_ring_modified(G, ring_size=10):
    start_idx = max(G.nodes) + 1
    fraud_ring = list(range(start_idx, start_idx + ring_size))
    G.add_nodes_from(fraud_ring)
    for i in fraud_ring:
        for j in fraud_ring:
            if i != j:
                G.add_edge(i, j)
    return G, fraud_ring

undirected_graph, fraud_ring = inject_fraud_ring_modified(undirected_graph)
```

Step 4: Convert to Directed Graph with Attributes

In actual transaction systems, data is directional, money flows from one account to another, rather than bidirectionally. To reflect this, I converted the undirected graph into a directed one. I also simulate two key edge attributes: transaction `weight`: The amount of money transferred and `timestamp`: the time of transaction. These additions bring us closer to the nature of real data, allowing for further analyses such as temporal trends and weighted centrality computations. Direction and attributes are critical in fraud detection as they help distinguish legitimate behavior from suspicious flows.

In [4]:
```
def to_directed_with_attributes(G):
    DG = nx.DiGraph()
    for u, v in G.edges():
        weight = np.random.uniform(10, 1000)
        timestamp = datetime.datetime.now() - datetime.timedelta(days=np.random.randint(0, 365))
        DG.add_edge(u, v, weight=weight, timestamp=timestamp)
    return DG

directed_graph = to_directed_with_attributes(undirected_graph)
```

Step 5: Compute Centrality and Risk Score

Graph centrality measures allow us to assess how 'important' each node is within the network structure. I computed four metrics: degree centrality (number of direct connections), betweenness centrality (how often a node lies on shortest paths between others), closeness centrality (average distance from a node to all others), and eigenvector centrality (influence based on connected neighbors). Each captures different behavioral nuances, some users might appear suspicious due to their excessive activity (high degree), while others act as intermediaries or control points (high betweenness). By combining these into a unified fraud risk score, I rank nodes according to their likelihood of engaging in abnormal or fraudulent behavior.

In [5]:
```
degree = nx.degree_centrality(directed_graph)
betweenness = nx.betweenness_centrality(directed_graph)
closeness = nx.closeness_centrality(directed_graph)
try:
    eigenvector = nx.eigenvector_centrality(directed_graph, max_iter=500)
except nx.PowerIterationFailedConvergence:
    eigenvector = {n: 0 for n in directed_graph.nodes()}

combined_risk_scores = {
    n: 0.25 * degree[n] + 0.25 * betweenness[n] + 0.25 * closeness[n] + 0.25 * eigenvector[n]
    for n in directed_graph.nodes()
}
nx.set_node_attributes(directed_graph, combined_risk_scores, 'risk_score')
```

Step 6: Community Detection

I apply label propagation, an unsupervised algorithm, to detect communities within the network. A community is a group of nodes that are more densely connected to each other than to the rest of the graph. In fraud analytics, these communities may correspond to social or operational groups, or in some cases, organized fraud rings. The detected groups help us segment the network into behavioral clusters, which is useful for understanding collective risk or collusion. If a whole community contains nodes with high fraud scores, that group may warrant deeper investigation.

In [6]:
```
communities = list(label_propagation_communities(directed_graph.to_undirected()))
community_map = {}
for i, comm in enumerate(communities):
    for node in comm:
        community_map[node] = i
nx.set_node_attributes(directed_graph, community_map, 'community')
```

Step 7: Summarize Risk Scores and Community Structure

With centrality and community data in place, I built a summary table for each node that includes its unique identifier, computed fraud risk score, and associated community. Sorting this table helps us identify high-risk entities for targeted reviews. This structured format enables downstream applications such as audit prioritization, alerts in a transaction monitoring system, or inputs to a machine learning classifier.

In [7]:
```
summary_df = pd.DataFrame({
    'Node': list(combined_risk_scores.keys()),
    'RiskScore': list(combined_risk_scores.values()),
    'Community': [community_map[n] for n in combined_risk_scores]
})
summary_df.sort_values(by='RiskScore', ascending=False).head(10)
```

Out[7]:

	Node	RiskScore	Community
299	293	0.122723	50
223	242	0.116122	16
283	296	0.114358	39
212	291	0.114096	44
234	292	0.113689	48
0	0	0.044498	0
254	208	0.043828	17
198	264	0.042598	17
1	1	0.037402	0
56	4	0.037319	0

Step 8: Community-Level Risk Summary

In addition to looking at individual risk scores, I compute summary statistics for each community: average risk, maximum risk, and the number of nodes. This helps us identify not just risky individuals but potentially collusive clusters. For instance, a small community with uniformly high risk scores may represent a fraud ring or shell entities involved in circular transactions. On the other hand, large communities with moderate risk may indicate systemic process issues or broader exposure. These insights support both forensic investigations and policy-level controls.

In [8]:
```
community_risk_summary = summary_df.groupby("Community").agg({
    "RiskScore": ["mean", "max", "count"]
}).sort_values(("RiskScore", "mean"), ascending=False)
community_risk_summary.columns = ["AvgRisk", "MaxRisk", "MemberCount"]
community_risk_summary.reset_index().head(10)
```

Out[8]:

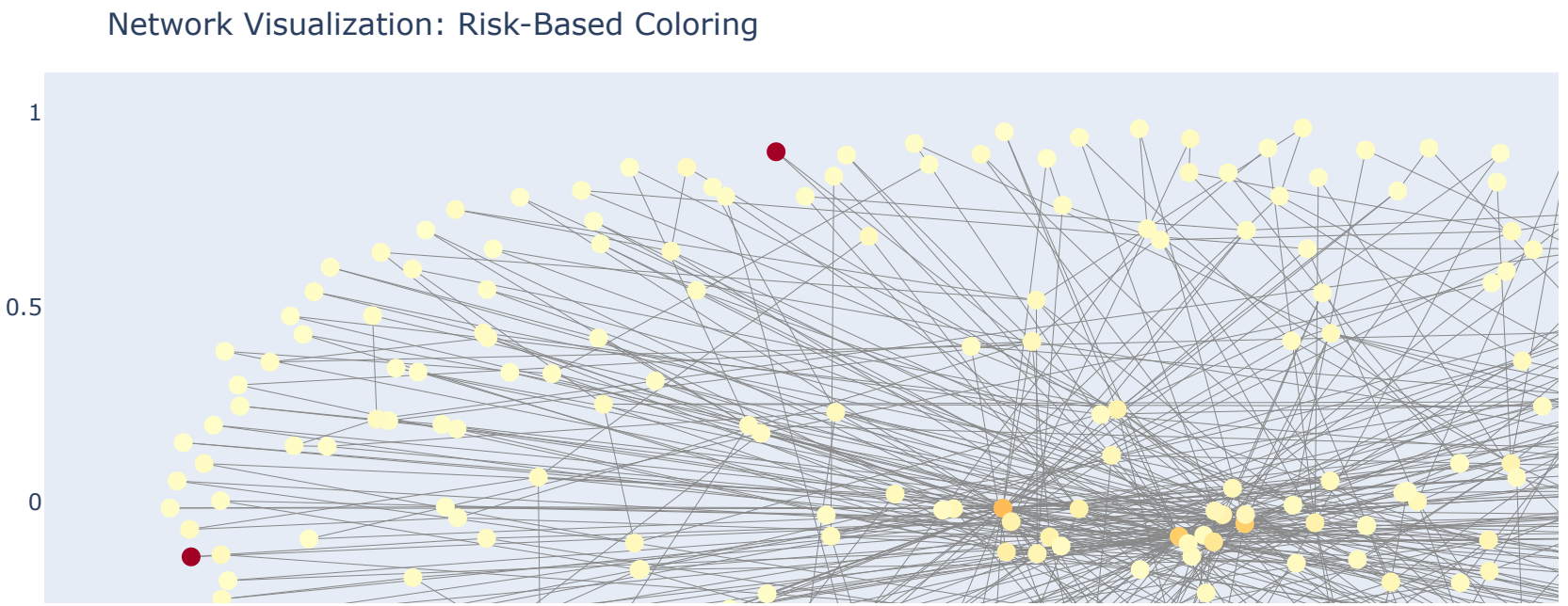
	Community	AvgRisk	MaxRisk	MemberCount
0	48	0.061013	0.113689	2
1	39	0.044660	0.114358	3
2	44	0.042926	0.114096	3
3	16	0.034305	0.116122	4
4	50	0.030679	0.122723	5
5	17	0.021671	0.043828	5
6	60	0.011009	0.015414	10
7	54	0.007897	0.009364	3
8	43	0.007688	0.007712	2
9	33	0.007590	0.013244	4

Step 9: Visualize the Graph

I plot the network using an interactive force-directed layout to show relationships and fraud risk visually. Nodes are placed using a spring layout algorithm where the spatial position reflects the structure of the network—connected nodes appear closer together. I

colored each node according to its risk score, allowing us to easily spot high-risk actors and potentially risky clusters. Interactive features like hover text provide details for deeper manual inspection. This kind of visualization is often used in operational settings to assist analysts with triage and anomaly exploration.

In [9]:
```
pos = nx.spring_layout(directed_graph, seed=42)
for n in directed_graph.nodes():
    directed_graph.nodes[n]['pos'] = pos[n]

edge_x, edge_y = [], []
for edge in directed_graph.edges():
    x0, y0 = directed_graph.nodes[edge[0]]['pos']
    x1, y1 = directed_graph.nodes[edge[1]]['pos']
    edge_x += [x0, x1, None]
    edge_y += [y0, y1, None]

edge_trace = go.Scatter(x=edge_x, y=edge_y, line=dict(width=0.5, color='#888'),
                        hoverinfo='none', mode='lines')

node_x, node_y, risk_color = [], [], []
for node in directed_graph.nodes():
    x, y = directed_graph.nodes[node]['pos']
    node_x.append(x)
    node_y.append(y)
    risk_color.append(directed_graph.nodes[node]['risk_score'])

node_trace = go.Scatter(x=node_x, y=node_y, mode='markers',
    marker=dict(size=10, color=risk_color, colorscale='YlOrRd', colorbar=dict(title="Risk Score")),
    text=[f'Node {n}<br>Risk: {combined_risk_scores[n]:.3f}' for n in directed_graph.nodes()],
    hoverinfo='text')

fig = go.Figure(data=[edge_trace, node_trace],
         layout=go.Layout(title='Network Visualization: Risk-Based Coloring',
                          titlefont_size=16, showlegend=False,
                          hovermode='closest', margin=dict(b=20,l=5,r=5,t=40),
                          xaxis=dict(showgrid=False, zeroline=False),
                          yaxis=dict(showgrid=False, zeroline=False)))
fig.show()
```



Step 10: Summary and Recommendations

This notebook demonstrated how to construct and analyze a synthetic transaction network for the purpose of fraud detection. Beginning with the generation of a scale-free graph, I introduced a collusive fraud ring, assigned realistic edge attributes, and converted the graph into a directed format to simulate transactional directionality. Centrality metrics—including degree, betweenness, closeness, and eigenvector—were computed to assess the prominence and structural influence of each node. I then combined these into a unified fraud risk score.

To detect organized fraud, I applied label propagation for community detection, allowing us to identify tightly connected clusters of nodes. A community-level risk analysis further helped in pinpointing groups with collectively high fraud potential.

What This Analysis Recommends:

- **Investigate nodes with the highest risk scores**: These often serve as transaction hubs or exhibit abnormal connectivity patterns.
- **Prioritize high-risk communities**: Groups of users with elevated risk suggest possible collusion.
- **Use risk scores as triage inputs**: Flag nodes or communities for further due diligence, audits, or enhanced monitoring.
- **Apply this model to real transaction logs**: With proper tuning, this graph-based approach can highlight fraud that traditional methods may miss.

Overall, this methodology supports proactive fraud detection and investigation by leveraging graph theory and network science.

In [ ]: