# Phase 3 TFF: Simulation Data and Testing Guide for Advanced Analysis

This guide provides details on generating simulation data and testing the Phase 3 Federated Learning (FL) system for Advanced Behavioral Analysis and Dispute Resolution in your ChainFLIP project. It builds upon the Phase 3 design document and the TFF framework.

## 1. Recap of Phase 3 Goals

- Detect Arbitrator Bias.
- Predict High-Risk Disputes.
- Identify Nodes Prone to Disputes.
- Advanced Anomaly Detection (e.g., time-series behavior).

## 2. Setting up for Phase 3 Simulation

As Phase 3 involves multiple distinct tasks, it's best to organize code into subdirectories within a main Phase 3 folder.

1. Inside `ChainFLIP_FL_Dev/`, create `tff_advanced_analysis/`.
2. Inside `tff_advanced_analysis/`, you might create further subdirectories like:
    - `arbitrator_bias/`
    - `dispute_risk/`
    - `node_behavior_timeseries/`
3. Each subdirectory would contain its own `data_preparation_p3_task.py`, `model_definition_p3_task.py`, and `run_simulation_p3_task.py`, adapted from the Phase 1/2 structure.

For this guide, we will provide conceptual data generation snippets that would go into these respective `data_preparation_p3_task.py` files.

## 3. Generating Synthetic Data for Phase 3 Tasks

### A. Arbitrator Bias Detection

- **File:** `ChainFLIP_FL_Dev/tff_advanced_analysis/arbitrator_bias/data_preparation_p3_arbitrator.py`
- **Features (Example - `NUM_P3_ARB_FEATURES = 5`):** `avg_vote_alignment_with_outcome`, `avg_vote_alignment_with_peers`, `disputes_participated`, `avg_value_disputes_favored_partyA`, `avg_value_disputes_favored_partyB`.
- **Label:** 0 for unbiased, 1 for biased.

```
# In data_preparation_p3_arbitrator.py
import tensorflow as tf
import numpy as np
import random

NUM_P3_ARB_FEATURES = 5
ELEMENT_SPEC_P3_ARB = (
    tf.TensorSpec(shape=(NUM_P3_ARB_FEATURES,), dtype=tf.float32),
    tf.TensorSpec(shape=(1,), dtype=tf.int32)
)

SIM_ARBITRATORS = [f"arb_{i}" for i in range(20)]
ARBITRATOR_PROFILES = {}
for i, arb_id in enumerate(SIM_ARBITRATORS):
    ARBITRATOR_PROFILES[arb_id] = {
        "is_biased": True if i % 4 == 0 else False, # ~25% are biased
        "bias_factor": random.uniform(0.6, 0.9) if (i % 4 == 0) else random.uniform(0.4,
0.6)
    }

def simulate_arbitrator_performance(arbitrator_id, num_disputes_arbitrated=50):
    profile = ARBITRATOR_PROFILES[arbitrator_id]
    label = [1] if profile["is_biased"] else [0]

    # Simulate features based on bias

# Feature 0: vote_alignment_with_outcome (biased might be lower if they vote against
fair outcomes)
    f0 = random.uniform(0.3, 0.7) if profile["is_biased"] else random.uniform(0.6,
0.95)
    # Feature 1: vote_alignment_with_peers (biased might deviate more)
    f1 = random.uniform(0.4, 0.7) if profile["is_biased"] else random.uniform(0.7, 0.9)
    # Feature 2: disputes_participated
    f2 = float(num_disputes_arbitrated)
    # Feature 3 & 4: Simplified bias towards favoring one party in value (e.g. party A vs B)
    # Assume higher value for party A if biased towards A
```

```
    favored_A_value = random.uniform(1000, 5000) * profile["bias_factor"]
    favored_B_value = random.uniform(1000, 5000) * (1.0 - profile["bias_factor"])
    f3 = favored_A_value
    f4 = favored_B_value

    features = np.array([f0, f1, f2, f3, f4], dtype=np.float32)
    return features, np.array(label, dtype=np.int32)

def load_local_data_for_p3_arbitrator_client(client_id: str, assigned_arbitrators:
list[str]):
    client_features = []
    client_labels = []
    for arb_id in assigned_arbitrators:
        features, label = simulate_arbitrator_performance(arb_id,
num_disputes_arbitrated=random.randint(20,100))
        client_features.append(features)
        client_labels.append(label)
    if not client_features: # Handle empty case
        return tf.data.Dataset.from_tensor_slices((
            np.zeros((0, NUM_P3_ARB_FEATURES), dtype=np.float32),
            np.zeros((0, 1), dtype=np.int32)
        ))
    return tf.data.Dataset.from_tensor_slices((np.array(client_features),
np.array(client_labels)))

def make_federated_data_p3_arbitrator(num_fl_clients=3):
    # Distribute arbitrators among FL clients
    random.shuffle(SIM_ARBITRATORS)
    arbitrators_per_fl_client = len(SIM_ARBITRATORS) // num_fl_clients
    client_datasets = []
    for i in range(num_fl_clients):
        start_idx = i * arbitrators_per_fl_client
        end_idx = (i+1) * arbitrators_per_fl_client if i < num_fl_clients -1 else
len(SIM_ARBITRATORS)

client_datasets.append(load_local_data_for_p3_arbitrator_client(f"fl_arb_client_{i}",
SIM_ARBITRATORS[start_idx:end_idx]))
    return client_datasets

# Add __main__ for testing this script independently
```

### B. High-Risk Dispute Prediction

- **File:** `ChainFLIP_FL_Dev/tff_advanced_analysis/dispute_risk/data_preparation_p3_dispute.py`
- **Features (Example - `NUM_P3_DR_FEATURES = 6`):** `dispute_value`, `num_parties_involved`, `avg_reputation_parties`, `reputation_diff_parties`, `num_evidence_items` (simulated), `prior_fl_risk_initiator`.
- **Label:** 0 for normal-risk, 1 for high-risk.

```
# In data_preparation_p3_dispute.py
import tensorflow as tf
import numpy as np
import random

NUM_P3_DR_FEATURES = 6
ELEMENT_SPEC_P3_DR = (
    tf.TensorSpec(shape=(NUM_P3_DR_FEATURES,), dtype=tf.float32),
    tf.TensorSpec(shape=(1,), dtype=np.int32)
)

def simulate_dispute_characteristics(dispute_id):
    is_high_risk = random.random() < 0.3 # ~30% are high-risk
    label = [1] if is_high_risk else [0]

    # Simulate features
    f0_value = random.uniform(100, 100000) * (1.5 if is_high_risk else 1.0) # Higher
value if high-risk
    f1_parties = random.randint(2,5)
    f2_avg_rep = random.uniform(10,100)
    f3_rep_diff = random.uniform(0, 50) * (1.2 if is_high_risk else 0.8)
    f4_evidence = random.randint(1,20) * (0.7 if is_high_risk else 1.3) # Less evidence
if high-risk (e.g. fraud)
    f5_prior_risk_initiator = random.uniform(0,1) * (1.8 if is_high_risk else 0.5)

    features = np.array([f0_value, f1_parties, f2_avg_rep, f3_rep_diff, f4_evidence,
f5_prior_risk_initiator], dtype=np.float32)
    return features, np.array(label, dtype=np.int32)

def load_local_data_for_p3_dispute_client(client_id: str,
num_disputes_to_generate=100):
    # Each client might observe/report a set of disputes
    client_features = []
    client_labels = []
    for i in range(num_disputes_to_generate):
        features, label = simulate_dispute_characteristics(f"dispute_{client_id}_{i}")
        client_features.append(features)
        client_labels.append(label)
    return tf.data.Dataset.from_tensor_slices((np.array(client_features),
np.array(client_labels)))

def make_federated_data_p3_dispute(num_fl_clients=3, disputes_per_client=100):
    return [load_local_data_for_p3_dispute_client(f"fl_disp_client_{i}",
disputes_per_client) for i in range(num_fl_clients)]

# Add __main__ for testing this script independently
```

### C. Time-Series Behavioral Anomaly Detection for Nodes

- **File:** `ChainFLIP_FL_Dev/tff_advanced_analysis/node_behavior_timeseries/data_preparation_p3_timeseries.py`
- **Features (Example - `NUM_P3_TS_FEATURES = 3` per timestep):** `tx_frequency_daily`, `avg_tx_value_daily`, `new_interactions_daily`.
- **Model Type:** Often unsupervised (e.g., Autoencoder, LSTM-Autoencoder). Label might be reconstruction error or a binary anomaly flag if using supervised anomaly detection.
- **Data Shape:** `(num_nodes, timesteps, features_per_timestep)`.

```python
# In data_preparation_p3_timeseries.py
import tensorflow as tf
import numpy as np
import random

NUM_P3_TS_FEATURES = 3
TIMESTEPS = 30 # e.g., 30 days of data

# For an autoencoder, labels are the same as inputs, or use reconstruction error for anomaly scoring
ELEMENT_SPEC_P3_TS = (
    tf.TensorSpec(shape=(TIMESTEPS, NUM_P3_TS_FEATURES), dtype=tf.float32), # Input sequence
    tf.TensorSpec(shape=(TIMESTEPS, NUM_P3_TS_FEATURES), dtype=tf.float32)  # Output sequence (for autoencoder)
)

SIM_NODES_TS = [f"ts_node_{i}" for i in range(50)]

def generate_node_timeseries(node_id):
    # Normal behavior
    base_tx_freq = random.uniform(5, 20)
    base_tx_val = random.uniform(100, 500)
    base_new_interact = random.uniform(1, 5)

    sequence = []
    is_anomalous_node = random.random() < 0.2 # 20% of nodes exhibit anomaly at some point
    anomaly_start_step = random.randint(TIMESTEPS // 2, TIMESTEPS - 5) if is_anomalous_node else TIMESTEPS

    for step in range(TIMESTEPS):
        is_anomaly_now = is_anomalous_node and step >= anomaly_start_step
        factor = 3.0 if is_anomaly_now else 1.0 # Anomaly makes values spike
        noise = np.random.normal(0, 0.1, NUM_P3_TS_FEATURES)

        f0 = max(0, base_tx_freq * factor * (1 + noise[0]))
        f1 = max(0, base_tx_val * (factor if random.random() > 0.5 else 1/factor) * (1 +
```

```
noise[1])) # Value might spike or drop
        f2 = max(0, base_new_interact * factor * (1 + noise[2]))
        sequence.append([f0, f1, f2])

    seq_array = np.array(sequence, dtype=np.float32)
    return seq_array, seq_array # Input and target are same for autoencoder

def load_local_data_for_p3_timeseries_client(client_id: str, assigned_nodes:
list[str]):
    client_input_seqs = []
    client_target_seqs = []
    for node_id in assigned_nodes:
        input_s, target_s = generate_node_timeseries(node_id)
        client_input_seqs.append(input_s)
        client_target_seqs.append(target_s)
    if not client_input_seqs: # Handle empty case
        return tf.data.Dataset.from_tensor_slices((
            np.zeros((0, TIMESTEPS, NUM_P3_TS_FEATURES), dtype=np.float32),
            np.zeros((0, TIMESTEPS, NUM_P3_TS_FEATURES), dtype=np.float32)
        ))
    return tf.data.Dataset.from_tensor_slices((np.array(client_input_seqs),
np.array(client_target_seqs)))

def make_federated_data_p3_timeseries(num_fl_clients=3):
    random.shuffle(SIM_NODES_TS)
    nodes_per_fl_client = len(SIM_NODES_TS) // num_fl_clients
    client_datasets = []
    for i in range(num_fl_clients):
        start_idx = i * nodes_per_fl_client
        end_idx = (i+1) * nodes_per_fl_client if i < num_fl_clients -1 else
len(SIM_NODES_TS)

client_datasets.append(load_local_data_for_p3_timeseries_client(f"fl_ts_client_{i}",
SIM_NODES_TS[start_idx:end_idx]))
    return client_datasets

# Add __main__ for testing this script independently
```

## 4. Adapting TFF Scripts for Phase 3 Tasks

For each task (Arbitrator Bias, Dispute Risk, Time-Series Anomaly): 1. **`model_definition_p3_task.py`:** * Define `NUM_P3_TASK_FEATURES`, `TIMESTEPS` (if applicable), and `ELEMENT_SPEC_P3_TASK`. * Create a Keras model suitable for the task: * Simple Dense network for Arbitrator Bias and Dispute Risk classification. * LSTM Autoencoder for Time-Series Anomaly Detection (input and output layers match `(TIMESTEPS, NUM_P3_TS_FEATURES)`). * Wrap it using `tff_model_fn`. 2. **`federated_training_p3_task.py`:** * Likely similar to Phase 1/2, using build_weighted_fed_avg . * For unsupervised autoencoders, the loss would be reconstruction loss (e.g., Mean Squared Error). 3. **run_simulation_p3_task.py :** * Import from the correct data_preparation_p3_task.py . * Use the corresponding make_federated_data_p3_task . * Batch client datasets appropriately ( preprocess_client_dataset function). * Interpret metrics: Accuracy/AUC for classification tasks. For autoencoders, monitor reconstruction loss; lower is better. Anomalies are detected by high reconstruction error on individual samples post-training.

## 5. Running and Testing Phase 3 Simulations

- For each task, navigate to its subdirectory (e.g., ChainFLIP_FL_Dev/tff_advanced_analysis/arbitrator_bias/ ).
- Run python run_simulation_p3_task.py .
- **Interpreting Results:**
  - **Arbitrator Bias/Dispute Risk:** Look for the model to achieve good accuracy/AUC in classifying biased arbitrators or high-risk disputes based on the simulated patterns.
  - **Time-Series Anomaly:** Monitor the reconstruction loss during training. After training, feed both normal and simulated anomalous sequences to the global autoencoder model. Anomalous sequences should have significantly higher reconstruction errors.

## 6. Next Steps for Phase 3

- **Deep Dive into Feature Engineering:** The simulated features here are illustrative. Real-world implementation requires careful selection and engineering of features from actual DisputeResolution.sol data and other sources.
- **Real Data Integration:** Plan how FL clients will programmatically access and process the necessary on-chain data.
- **Admin Dashboard:** Design how insights from these advanced models (e.g., arbitrator bias scores, dispute risk levels, node anomaly alerts) will be presented to and utilized by administrators.

This guide provides a foundation for simulating and testing the advanced FL capabilities envisioned for Phase 3. Each task is a mini-project in itself, requiring careful data handling and model selection.