Statistical Comparison of PoS Committee Selection Algorithms

Robert Jones

23 April 2025

1 Simulation Experiment

We describe a Monte Carlo simulation experiment designed to evaluate two algorithms (A and B) that select committee members in a Proof-of-Stake (PoS) blockchain to achieve Byzantine Fault Tolerance (BFT) consensus. Their performance is measured by their robustness against faults when maintaining the 2/3 majority required for consensus. In this experiment, we sample a set of system configuration parameters α from a joint distribution. For each sampled α, we run many epochs (committee selections) for both algorithms and estimate a *fault tolerance score* (FTS) for both algorithms. The results are then analyzed using statistical tests to determine if there is a significant difference in FTS between the two algorithms.

For a given algorithm, let

$$p_f = \text{Probability that the system tolerates } f \text{ faults}$$

for $f = 1, 2, 3, \ldots$. The **Fault Tolerance Score (FTS)** for the algorithm is then defined as:

$$\text{FTS} = \sum_{f=1}^{\infty} p_f,$$

which can be interpreted as the expected number of faults the system can tolerate before losing consensus.

- For **Algorithm A**, we denote the score as FTS_A.
- For **Algorithm B**, we denote the score as FTS_B.

A higher Fault Tolerance Score indicates that an algorithm can, on average, tolerate more faults while still preserving its 2/3 majority, thereby demonstrating better resilience.

To compare the algorithms, we examine the difference in their scores:

$$\Delta\text{FTS} = \text{FTS}_B - \text{FTS}_A.$$

- If $\Delta\text{FTS} > 0$, then **Algorithm B** is more resilient (i.e., it tolerates a higher number of faults on average).
- If $\Delta\text{FTS} < 0$, then **Algorithm A** performs better in terms of fault tolerance.
- If $\Delta\text{FTS} \approx 0$, then there is no significant difference between the two in terms of their fault tolerance capability.

2 Paired Statistical Testing

A crucial aspect of this experiment is the variation introduced by sampling α from a distribution. By pairing the outcomes for A and B within each α, we have structured the test as a randomized block design, where each block is a fixed environment α and we compare two treatments (A vs B) within that block. This design improves sensitivity by removing inter-block (inter-α) variability. Essentially, each scenario acts as its own control. If we instead ignored α and pooled all epochs together, random differences in scenario difficulty would obscure the comparison.

Given that we have a sample of paired differences $\Delta\text{FTS}(\alpha)$ for many random scenarios, the natural approach uses a paired statistical test to assess whether the mean of $\Delta\text{FTS}(\alpha)$ is more significant than zero. The paired test treats the set of differences as a single sample. If the number of sampled scenarios is large enough, one common choice is the paired Student's t-test, which tests whether the mean difference is significantly different from zero. The paired t-test assumes the differences $\Delta\text{FTS}(\alpha)$ are approximately normally distributed (an assumption often reasonable by the Central Limit Theorem if the number of samples is moderately large). It also assumes the sample of scenarios is independent. Here we know each α is drawn independently, but we cannot assess normality in the data distribution.

2.1 Hypothesis Test

To evaluate whether Algorithm B is statistically superior to Algorithm A, we conduct a *one-sided hypothesis test*:

- **Null Hypothesis**, H_0: Algorithm B offers no improvement over A.

$$\text{mean}(\Delta\text{FTS}) = 0$$

- **Alternative Hypothesis**, H_1: Algorithm B is more fault tolerant.

$$\text{mean}(\Delta\text{FTS}) > 0$$

2.2 Bootstrap Sampling

Traditional statistical tests assume normality in the data distribution, which, after some testing, we determine is not the case here—rather, the data distribution is heavy tailed. Therefore, we will use a non-parametric alternative called *bootstrap sampling*. This approach is particularly useful when the underlying distribution is unknown or when the sample size is small. The bootstrap method allows us to estimate the sampling distribution of a statistic —in this case, $\Delta\text{FTS}(\alpha)$ for a given parameter set α— by resampling the observed data *with replacement*.

Now, we can conduct bootstrap resampling as follows:

1. **Resample**: Draw a large number (e.g., 10,000) of bootstrap samples from the centered differences (with replacement).
2. **Statistic**: For each bootstrap sample, compute its mean. This collection of sample means forms an empirical null distribution of the mean difference.
3. **Construct Empirical Null Distribution**: Shift the observed differences by subtracting the empirical mean (centered data) to simulate what would be expected under H_0.

4. **Compute Bootstrap Means**: For each bootstrap sample, calculate the mean difference in FTS scores.
5. **Formulate the One-Sided p-Value**: The one-sided p-value is calculated as

$$p = \frac{\text{number of bootstrap replicates with mean } \geq \text{observed mean}}{\text{total number of replicates}}.$$

6. **Interpret the p-value**: If this p-value is small (say, less than 0.05), it implies that such a high mean difference is very unlikely under the null hypothesis, lending support to the claim that algorithm B's performance is significantly better than algorithm A's.
7. **Conclusion**: If the p-value is small, we reject the null hypothesis H_0 in favor of the alternative hypothesis H_1. This suggests that Algorithm B is statistically significantly better than Algorithm A.

3 Experimental Results

Appenndix 1 contains the Python code used to conduct the Monte Carlo simulation experiment. The code implements the algorithms, generates the parameter configurations, and computes the Fault Tolerance Scores (FTS) for both Algorithm A and Algorithm B. It also includes the bootstrap sampling procedure to assess the statistical significance of the differences in FTS scores.

Running the code produces the results of the Monte Carlo simulation experiment, including the histograms of FTS scores for both algorithms and the histogram of the differences in FTS scores (ΔFTS). The code also performs the bootstrap hypothesis test to determine whether Algorithm B outperforms Algorithm A.

The output of the code:

```
simulation trials: 100% [1000/1000] [13:54<00:00,  1.20it/s]
Observed Mean Difference (B - A): 1.733
One-sided p-value: 0.0000
Conclusion: Algorithm B is significantly better than Algorithm A.
```

3.1 Discussion

The simulation trials were run for 1000 iterations, and the observed mean difference in Fault Tolerance Scores (FTS) between Algorithm B and Algorithm A was found to be 1.733. The one-sided p-value was less than 0.0001, indicating strong evidence against the null hypothesis H_0. Therefore, we conclude that Algorithm B is significantly better than Algorithm A in terms of fault tolerance.

The code also generates visualizations of the results—figures 1, 2, and 3— including histograms of the FTS scores for both algorithms and the histogram of the differences in FTS scores (ΔFTS). The histograms provide a visual representation of the distribution of FTS values for both algorithms across multiple parameter configurations.

The figures illustrate the results of our Monte Carlo simulation experiment. Figure 1 shows the comparison of Fault Tolerance Scores (FTS) between Algorithm A and Algorithm B across different parameter configurations. The overlayed histograms in figure 2 displays the distribution of FTS

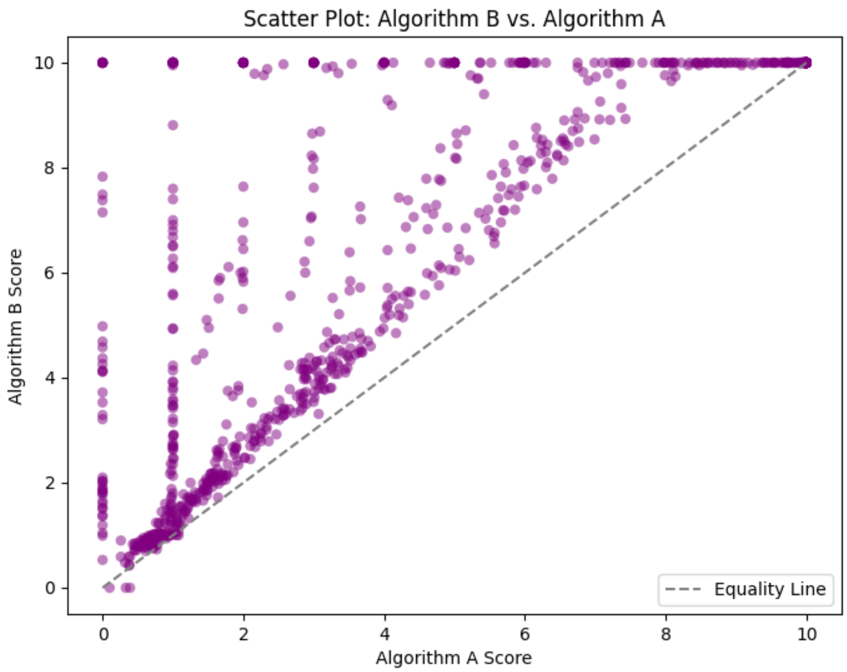

Figure 1: Comparison of Fault Tolerance Scores (FTS) between Algorithm A and Algorithm B across different parameter configurations. Higher scores indicate better fault tolerance.

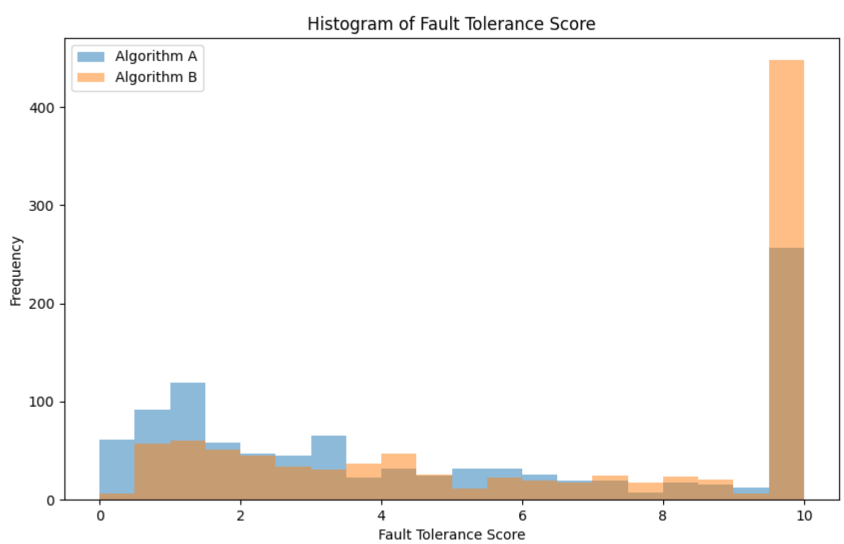

Figure 2: Comparison of FTS histograms between Algorithm A and Algorithm B. The histogram shows the distribution of FTS values for both algorithms across multiple parameter configurations.

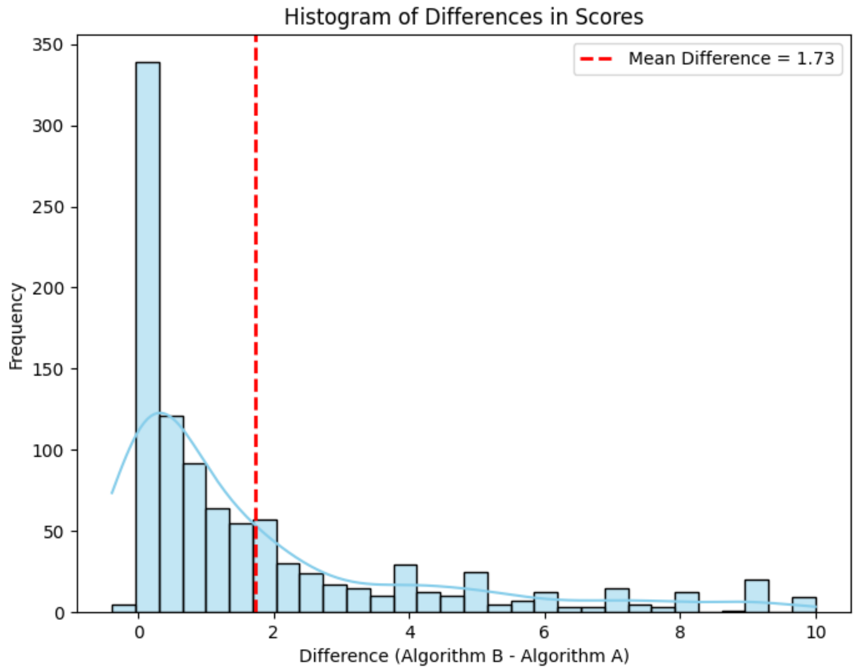

Figure 3: Histogram of the diffference in FTS scores (ΔFTS) between Algorithm A and Algorithm B.

values for both algorithms. Finally, figure 3 shows the histogram of the difference in FTS scores (ΔFTS) between Algorithm A and Algorithm B. The mean of the ΔFTS distribution is significantly greater than zero, with a p-value less than 0.05 from the bootstrap hypothesis test. This indicates that we can reject the null hypothesis H_0 in favor of the alternative hypothesis H_1, indicating that Algorithm B consistently outperforms Algorithm A in terms of fault tolerance.

4 Conclusion

Our experimental results show the distribution of ΔFTS values across multiple parameter configurations. As shown in figures, Algorithm B demonstrates consistently higher fault tolerance scores compared to Algorithm A. The bootstrap hypothesis test confirms that the observed differences are statistically significant, with a p-value less than 0.05. This suggests that Algorithm B is indeed more resilient in terms of fault tolerance, making it a preferable choice for PoS committee selection in Byzantine Fault Tolerance scenarios. The use of bootstrap sampling allowed us to overcome the limitations of traditional statistical tests, particularly in the presence of non-normal data distributions. By resampling the observed data and constructing an empirical null distribution, we were able to accurately assess the significance of the differences in FTS scores between the two algorithms. The results of this study provide valuable insights into the performance of PoS committee selection algorithms and highlight the importance of robust statistical methods in evaluating their effectiveness. Future work may involve further exploration of different parameter configurations, as well as the application of these methods to other PoS consensus algorithms.

A Appendix: Code

```
# !/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np
import pandas as pd
import scipy.stats as stats
import seaborn as sns
from matplotlib import pyplot as plt
from numpy import ceil, ndarray, ones, random, zeros
from tqdm import tqdm

from slot_alloc_sim import simulate_epoch_permissioned,
simulate_epoch_registered


def sample_multinomial(num_participants: int) -> ndarray:
    """This function generates a probability vector from a flat Dirichlet
distribution
    (with equal pseudo-counts for every participant)

    Args:
        num_participants: Number of participants (candidates from group)

    Returns:
        probabilities: A probability vector sampled from the Dirichlet
distribution
    """
    # Define a "flat" Dirichlet prior where every pseudo-count is 1.
    alpha = ones(num_participants)

    # Sample a probability vector from the Dirichlet distribution,
    # which represents the multinomial probabilities for each participant.
    return random.dirichlet(alpha)


def random_factor(n):
    """
    Computes all integer factors of a number and randomly selects one.

    Args:
        n (int): The number to find factors for

    Returns:
        int: A randomly selected factor of n
    """
    # Find all factors of n
    factors = []
    for i in range(1, int(n ** 0.5) + 1):
        if n % i == 0:
            factors.append(i)
            if i != n // i:  # Avoid adding the same factor twice for perfect
squares
                factors.append(n // i)

    # If no factors are found (should only happen if n=0), return 1
    if not factors:
        return 1
```

```
    # Return a random factor
    return random.choice(factors)


# Define the function to sample alpha from a joint distribution
def sample_alpha() -> dict:
    """
    Generates a sample allocation of seats among participants using a
multinomial
    distribution.

    The function determines the total number of seats and the number of
    participants from predefined options. It then generates probabilities for
    each participant and uses these probabilities to allocate seats
    proportionally. The result is returned as a dictionary containing all the
    generated values.

    Returns:
        dict: A dictionary containing the following keys and values:
            - total_seats (int): Total number of available seats.
            - num_participants (int): Number of participants.
            - p (List[float]): Probabilities for each participant.
            - seat_counts (List[int]): List containing the number of seats
              allocated to each participant.

    Raises:
        AssertionError: If the internal logic fails, such as the sum of seat
        counts exceeding the specified total seats due to an unexpected error.
    """
    # Consider a random number of registered seats over this range
    num_registered = random.choice([10, 20, 40, 80, 160, 320])

    # Let num_permitted seats be a random percentage of num_registered
    num_permitted = int(ceil(random.choice([0.1, 0.3, 0.5, 0.7]) *
num_registered))

    # Choose a random factor for seats_per_federated
    seats_per_federated = random_factor(num_permitted)

    # This is guaranteed now to be an integer number of federated participants
    num_federated = num_permitted // seats_per_federated

    total_seats = num_registered + num_permitted

    registered_probabilities = sample_multinomial(num_registered)

    max_faults = 10

    sample = dict(
        total_seats=total_seats,
        num_registered=num_registered,
        num_permitted=num_permitted,
        num_federated=num_federated,
        seats_per_federated=seats_per_federated,
        registered_probabilities=registered_probabilities,
        max_faults=max_faults,
        num_epochs=100,
        )
```

```
    return sample


def faults_tolerated(committee_seats: pd.Series) -> int:
    """
    Compute the number of faults tolerated by the committee

    Args:
        committee_seats (pd.Series): Series of committee seats.

    Returns:
        int: The number of faults tolerated by the committee.
    """
    voting_strength = committee_seats.sort_values(ascending=False).divide(
        committee_seats.sum(),
        )
    threshold = 1 / 3  # BFT finality risk threshold
    faults = np.where(np.cumsum(voting_strength) > threshold)[0][0]
    return faults


def simulate_epoch_federated(
    registered_seat_counts: ndarray,
    num_federated: int,
    seats_per_federated: int,
    verbose: bool = False,
    ) -> pd.DataFrame:
    """
    Creates a committee with registered and federated seats.

    The `simulate_epoch_federated` function allocates committee seats between two types of nodes: registered nodes
    (whose seat allocation is proportional to their stake) and federated nodes (which receive a fixed number of
    seats). It calculates the voting strength of each group, ensures they sum to 1.0, and returns a structured
    DataFrame containing the committee composition with seat assignments for both node types.

    Args:
        registered_seat_counts (ndarray): seat counts for registered participants.
        num_federated (int): Number of federated nodes.
        seats_per_federated (int): Number of seats per federated node.
        verbose (bool): Whether to print detailed information.
            Default is False.

    Returns:
        pd.DataFrame: Committee seats information with kind and seat assignments.
    """
    # Cast the seat counts to a pandas Series
    registered_seat_counts = pd.Series(registered_seat_counts, name="registered seats")

    # Get the number of registered participants (SPOs)
    num_registered = registered_seat_counts.shape[0]

    # Calculate the number of federated seats in the committee
```

```python
    federated_seats = seats_per_federated * num_federated

    # Committee size is the total number of registered and federated seats
    registered_seats = registered_seat_counts.sum()
    committee_size = registered_seats + federated_seats

    # Calculate the voting strength of the registered seats
    registered_voting_strength = registered_seats / committee_size

    # Calculate the voting strength of the federated seats
    federated_voting_strength = federated_seats / committee_size

    # Assert that the voting strengths sum to 1.0
    assert (
        registered_voting_strength + federated_voting_strength == 1.0
    ), "Voting strength does not sum to 1.0"

    # Create a series for the federated seats on the committee
    federated_seat_counts = pd.Series(
        ones(num_federated, dtype="int64") * seats_per_federated,
        index=[str(i) for i in range(num_registered + 1, num_registered + num_federated + 1)],
        dtype="int64",
        name="federated seats",
        )

    # Combine the federated and registered seats into a single DataFrame
    committee_seats = (
        pd.concat(
            [federated_seat_counts, registered_seat_counts],
            keys=["federated", "registered"],
            names=["kind", "index"],
            ignore_index=False,
            )
        .reset_index()
        .rename(columns={0: "seats"})
        .set_index("index")
        .sort_values(by=["seats", "kind"], ascending=[False, True])
    )
    if verbose:
        print(
            f"Committee size .... = {committee_size} \n"
            "________________________________________\n"
            "Registered:\n"
            f"Number registered.. = {num_registered} \n"
            f"Number of seats.... = {registered_seats} \n"
            f"Voting strength.... = {registered_voting_strength:.2%} \n"
            "________________________________________\n"
            "Federated:\n"
            f"Number federated... = {num_federated} \n"
            f"Seats per federated = {seats_per_federated} \n"
            f"Number of seats.... = {federated_seats} \n"
            f"Voting strength.... = {federated_voting_strength:.2%}\n",
            )
    return committee_seats


def simulate_committee_federated(
    registered_probabilities: ndarray,
```

```python
    num_registered: int,
    num_federated: int,
    seats_per_federated: int,
    num_epochs: int,
    **kwargs,
    ) -> pd.DataFrame:
    """
    Simulate the allocation of committee seats over multiple epochs.

    This function simulates the process of assigning committee seats for a specified
    number of epochs using the simulate_epoch_federated function. It collects seat
    assignments across all epochs and returns them as a DataFrame.

    Args:
        registered_probabilities (ndarray): Probabilities for registered participants.
        num_registered (int): Number of registered participants.
        num_federated (int): Number of federated nodes.
        seats_per_federated (int): Number of seats per federated node.
        num_epochs (int): Number of epochs to simulate.
        kwargs: Additional keyword arguments.

    Returns:
        pd.DataFrame: DataFrame containing committee seat assignments for all epochs,
                      with epochs as rows and committee members as columns.
    """
    committee_list = []
    for epoch in range(num_epochs):
        # Sample the committee seat configuration from the multinomial distribution
        # parameterized by registered_probabilities.
        registered_seat_counts = random.multinomial(num_registered, registered_probabilities)

        # Assign committee seats for the current epoch
        committee = simulate_epoch_federated(
            registered_seat_counts=registered_seat_counts,
            num_federated=num_federated,
            seats_per_federated=seats_per_federated,
            )
        committee_list.append(committee.seats)

    assert len(committee_list) == num_epochs, "Number of epochs does not match."

    committee_seats = pd.concat(committee_list, keys=range(num_epochs), axis=1).T

    return committee_seats


def simulate_proposed(
    registered_probabilities: ndarray,
    num_registered: int,
    num_permitted: int,
    num_epochs: int,
```

```python
    **kwargs,
    ) -> pd.DataFrame:
    """Runs the proposed algorithm simulation. For each epoch, simulates both the
    registered and permissioned candidate selection. Stores the number of slots
    each candidate gets and returns the cumulative results.

    Args:
        registered_probabilities (ndarray): Probabilities for registered candidates.
        num_registered (int): Number of registered candidates.
        num_permitted (int): Number of permitted (federated)nodes.
        num_epochs (int): Number of epochs to simulate.
        kwargs: Additional keyword arguments.

    Returns:
        pd.DataFrame: DataFrame containing committee seat assignments for all epochs,
            with epochs as rows and committee members as columns.
    """
    permissioned_candidates = [f"P{i}" for i in range(num_permitted)]
    permissioned_results = {name: [] for name in permissioned_candidates}
    registered_results = {i: [] for i in range(num_registered)}

    for _ in range(num_epochs):
        registered_candidates = {i: registered_probabilities[i] for i in registered_results.keys()}

        # Simulate this epoch
        reg_assign = simulate_epoch_registered(registered_candidates, num_registered)
        perm_assign = simulate_epoch_permissioned(permissioned_candidates, num_permitted)

        # Collect results
        for name in registered_candidates:
            registered_results[name].append(reg_assign[name])
        for name in permissioned_candidates:
            permissioned_results[name].append(perm_assign[name])

    # Combine the two dictionaries
    combined = registered_results.copy()
    combined.update(permissioned_results)

    # Convert to DataFrame
    committee_seats = (
        pd.DataFrame.from_dict(combined, orient="index")
        .astype(int)
        .fillna(0)
        .transpose()
    )
    return committee_seats


def calculate_fault_tolerance_probability(
    committee_seats: pd.DataFrame,
    fault_tolerance: int = 1,
    ) -> float:
```

```python
    """
    Calculate the probability of tolerating a given number of faults
    in a committee of a given size.
    The function uses Monte Carlo simulation to estimate the probability.
    Args:
        committee_seats (pd.DataFrame): DataFrame of committee seat
            assignments of both registered and permissioned members
            for each epoch.
        fault_tolerance (int): Number of faults to tolerate.

    Returns:
        float: Estimated probability of tolerating the given number of faults.
    """
    if fault_tolerance == 0:
        probability = 1.0
    else:
        # Calculate success rate through Monte Carlo simulation
        probability = (
            committee_seats.apply(faults_tolerated, axis=1) >= fault_tolerance
        ).mean()
    return probability


# Stub functions for Algorithm A and B
def algorithm(function, **params) -> float:
    """
    Computes fault tolerance probabilities using an algorithm that processes a defined function and its
    parameters to determine committee seats and calculate probabilities for a fault tolerance metric.

    Args:
        function (Callable): A function that determines committee seats based on provided parameters.
        **params: A set of keyword arguments to be passed into the function `function`. Must include
            'max_faults' which defines the maximum faults for calculating probabilities; defaults to 5
            if not specified.

    Returns:
        mean_fault_tolerance: (float) The mean fault tolerance probability for the algorithm.
    """
    committee_seats = function(**params)

    # Calculate fault tolerance probabilities for the algorithm
    faults = np.arange(1, params.get("max_faults", 5) + 1)
    ftprob = pd.Series(0.0, index=faults, name="probability")
    for f in faults:
        # p is the probability of tolerating f faults
        p = calculate_fault_tolerance_probability(committee_seats, fault_tolerance=f)
        ftprob.loc[f] = p
        if p == 0:  # since the rest of the series will be zero as well
            break

    # Compute the performance score as the sum of probabilities
    score = ftprob.sum()
```



```python

        return score


    def hypothesis_test(
        results_a: ndarray,
        results_b: ndarray,
        kind: str = "bootstrap",
        n_bootstrap: int = 10000,
        ):
        """
        Conducts a hypothesis test to compare two result sets, ‘results_a‘ and ‘results_b‘, based on the specified
        kind of statistical testing approach. By default, a bootstrap approach is used to calculate the one-sided
        p-value for the null hypothesis. Alternatively, a paired t-test can be performed. The results help determine
        whether algorithm B (corresponding to ‘results_b‘) is significantly better than algorithm A (corresponding
        to ‘results_a‘).

        Args:
            results_a: An array of numerical results for algorithm A.
            results_b: An array of numerical results for algorithm B.
            kind: The kind of statistical test to perform. Options are "bootstrap" (default) or "paired_t".
            n_bootstrap: The number of bootstrap resampling iterations to perform. Only applicable
                if the ‘kind‘ is "bootstrap".

        Raises:
            ValueError: If ‘kind‘ is not "bootstrap" or "paired_t".
            RuntimeError: If inconsistencies in data prevent calculations, such as mismatched array lengths
                or insufficient data.
        """
        if kind == "bootstrap":
            # Calculate the observed mean difference.
            differences = results_b - results_a
            obs_diff = np.mean(differences)

            # For a one-sided test (B > A), if the observed mean difference is non-positive,
            # the p-value would be 1 (or you might decide not to proceed).
            if obs_diff <= 0:
                p_value = 1.0
            else:
                # Center the differences: adjust the data so that its mean becomes 0
                # (the null hypothesis condition)
                differences_centered = differences - obs_diff

                # Bootstrap resampling from the centered differences.
                bootstrap_means = np.empty(n_bootstrap)
                for i in range(n_bootstrap):
                    sample = np.random.choice(differences_centered, size=len(differences_centered), replace=True)
                    bootstrap_means[i] = np.mean(sample)

```

```python
            # One-sided p-value: proportion of bootstrap means greater than or equal to the observed difference.
            p_value = np.mean(bootstrap_means >= obs_diff)

        print(f"Observed Mean Difference (B - A): {obs_diff:.3f}")
        print(f"One-sided p-value: {p_value:.4f}")

    else:
        # Perform a paired t-test to determine if B is significantly better than A
        t_stat, p_value = stats.ttest_rel(results_a, results_b, alternative='less')

        print(f"Mean mA: {results_a.mean():.3f}, Mean mB: {results_b.mean():.3f}")
        print(f"Paired t-test result: t-statistic = {t_stat:.3f}, p-value = {p_value:.3f}")

    if p_value < 0.05:
        print("Conclusion: Algorithm B is significantly better than Algorithm A.")
    else:
        print("Conclusion: No statistically significant difference between A and B.")


def monte_carlo_simulation(num_trials: int = 100) -> tuple[np.ndarray, np.ndarray]:
    """
    Run the Monte Carlo simulation to compare Algorithms A and B across random scenarios.

    Args:
        num_trials: (int) The number of trials to run the simulation for. Defaults to 50.

    Returns:
        tuple[ndarray[Any, float], ndarray[Any, float]]: results_a, results_b
    """
    a = simulate_committee_federated
    b = simulate_proposed

    results_a = zeros(num_trials)
    results_b = zeros(num_trials)

    for i in tqdm(range(num_trials), desc="simulation trials"):
        alpha = sample_alpha()

        results_a[i] = algorithm(a, **alpha)
        results_b[i] = algorithm(b, **alpha)

    hypothesis_test(results_a, results_b, kind="bootstrap")

    return results_a, results_b


def visualize_algorithm_comparison(results_a, results_b, show_plots=True):
    """
    Visualize the comparison between two algorithms through various plots.
```

```

    This function creates three visualization plots:
    1. Histogram of scores for both algorithms
    2. Histogram of differences between algorithm scores
    3. Scatter plot of paired algorithm results

    Args:
        results_a (ndarray): Array of scores from Algorithm A
        results_b (ndarray): Array of scores from Algorithm B
        show_plots (bool): Whether to display the plots immediately. Default is True.

    Returns:
        tuple: Three matplotlib figure objects (hist_fig, diff_fig, scatter_fig)
    """
    # Calculate differences
    differences = results_b - results_a  # difference: B - A

    # ------------------------------
    # 1. Scatter Plot of Paired Data
    # ------------------------------
    scatter_fig = plt.figure(figsize=(8, 6))
    plt.scatter(results_a, results_b, alpha=0.5, color='purple', edgecolor='none')
    plt.xlabel("Algorithm A Score")
    plt.ylabel("Algorithm B Score")
    plt.title("Scatter Plot: Algorithm B vs. Algorithm A")
    plt.plot(
        [min(results_a), max(results_a)], [min(results_a), max(results_a)],
        linestyle='--', color='gray', label='Equality Line',
        )
    plt.legend()

    # --------------------------
    # 2. Histogram each Score
    # --------------------------
    hist_fig = plt.figure(figsize=(10, 6))
    plt.hist(results_a, bins=20, label="Algorithm A", alpha=0.5)
    plt.hist(results_b, bins=20, label="Algorithm B", alpha=0.5)
    plt.xlabel("Fault Tolerance Score")
    plt.ylabel("Frequency")
    plt.title("Histogram of Fault Tolerance Score")
    plt.legend()

    # --------------------------
    # 3. Histogram of Differences
    # --------------------------
    diff_fig = plt.figure(figsize=(8, 6))
    sns.histplot(differences, bins=30, kde=True, color='skyblue', edgecolor='black')
    plt.xlabel("Difference (Algorithm B - Algorithm A)")
    plt.ylabel("Frequency")
    plt.title("Histogram of Differences in Scores")
    plt.axvline(
        np.mean(differences), color='red', linestyle='dashed', linewidth=2,
        label=f'Mean Difference = {np.mean(differences):.2f}',
        )
    plt.legend()
```

```python
        if show_plots:
            plt.show()

        return hist_fig, diff_fig, scatter_fig


if __name__ == "__main__":
    # Run the simulation
    results_a, results_b = monte_carlo_simulation(num_trials=1000)

    visualize_algorithm_comparison(results_a, results_b)
```

Listing 1: Python code for the Monte Carlo simulation experiment