# EigenLayer's Restaking Model: Expanding Capital Efficiency or Undermining Ethereum's Security?

By Benjamin Erickson

Abstract

*EigenLayer is an innovative protocol on Ethereum that lets validators reuse their staked ETH to help secure other decentralized services like oracles, bridges, and rollups. In return, validators can earn extra staking rewards. When Ethereum transitioned to Proof-of-Stake in 2022, it created a situation where billions of dollars in staked ETH became locked and unable to serve other economic purposes. EigenLayer's restaking model addresses that weakness and significantly improves capital efficiency for Ethereum validators. However, it also presents risks that could undermine the system. This paper argues that EigenLayer represents a genuine advancement in crypto-economic efficiency that, when properly governed, enhances rather than undermines Ethereum's security model.*

*To support this thesis, the analysis explores the tension between increased capital utilization and system security that EigenLayer presents. Using a custom simulation of 1,000 validators across four correlated failure scenarios, the analysis demonstrates that while restaking introduces systemic risks such as cascading penalties and validator overexposure, which are similar to rehypothecation in traditional finance, these risks remain manageable with proper safeguards. The research evaluates EigenLayer's existing protections and proposes additional measures including adaptive risk scoring and tiered slashing mechanisms. This analysis concludes that with strong governance, restaking can expand Ethereum's economic utility without compromising its foundational security.*

## I. Introduction

In blockchain networks, security and decentralization are often achieved through economic incentives tied directly to the use and protection of digital assets. Ethereum, the second-largest blockchain by market capitalization, transitioned from Proof-of-Work (PoW) to Proof-of-Stake (PoS) in 2022 to improve energy efficiency and manage increasing transaction volume. Under PoS, validators secure the Ethereum network by staking 32 ETH as collateral and running verification software to confirm the accuracy of new transactions before these transactions are added to the blockchain. In exchange for honest behavior, validators earn rewards of 3–5% annually. However, dishonest behavior or critical mistakes can result in slashing penalties, where a portion of the staked ETH is forfeited. By early 2025, over 34 million ETH, worth tens of billions of dollars, was staked, reflecting widespread confidence in Ethereum's decentralized security model.[1]

A limitation of Ethereum's PoS system is that once ETH is staked, it cannot be used for other productive purposes. This inefficiency creates opportunity costs, limiting the rewards earned by validators and requiring each new decentralized application (DApp) or protocol to establish its own set of validators to secure its network. This process is very costly, redundant, and an inefficient use of Ethereum's trusted network of validators.

EigenLayer, an innovative middleware protocol that connects Ethereum to new DApps, introduces a new restaking model to solve this problem. This process allows existing Ethereum validators and other operators to reuse staked ETH to secure additional decentralized services, called Actively Validated Services (AVSs). By accepting additional slashing conditions tied to these services, validators can earn incremental rewards without supplying new capital. This shared security model allows new protocols to tap into Ethereum's established network of validators, reducing the barrier to entry for new DApps while increasing the reward potential for validators.[2]

The restaking model has led some analysts to describe EigenLayer as the "fourth paradigm in crypto-economic capital efficiency," a new effort to solve the inefficient use of staked assets in decentralized finance (DeFi). It builds on earlier paradigms: 1) pure Proof-of-Work with no staking (Bitcoin), 2) direct staking with locked capital (Ethereum PoS), 3) liquid staking tokens that represent staked assets (Lido stETH), and 4) reusing staked assets to secure multiple protocols simultaneously (EigenLayer). [3]

[1] Beaconcha.in. (2025). *Ethereum validator data explorer*. https://beaconcha.in

[2] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

[3] McKinney, J. (2023). *EigenLayer explained: The fourth paradigm in crypto-economic capital efficiency*. https://www.youtube.com/watch?v=iMFscq9Sxdk

Unlike traditional PoS and liquid staking models that came before it, EigenLayer reuses staked assets to secure multiple protocols. This represents a significant leap in capital utilization and efficiency, transforming Ethereum's validator base into a security marketplace.

Despite these advancements, EigenLayer introduces notable risks. The most concerning is cascading slashing, where a failure in one AVS could trigger slashing penalties across all AVSs that share the same set of validators. As later discussed in Section III, this structure introduces systemic risks similar to traditional financial practices. Validators who overextend their collateral across too many AVSs may face multiple penalties, leading to forced exits and security disruptions. Additionally, AVSs that offer higher rewards may attract more validators, increasing the risk of centralization, and control in the hands of too few. Governance complexity also rises as EigenLayer introduces multiple stakeholders with overlapping interests and slashing policies.[4]

This paper examines whether EigenLayer's restaking improves capital efficiency without compromising Ethereum's security. First, it explains how EigenLayer's restaking model works, including validator participation, AVS integration, and the shared security concept. Next, it analyzes systemic risks and governance challenges, drawing on comparisons to traditional PoS and liquid staking models and financial practices. A custom simulation evaluates the impact of cascading slashing under stressful conditions. Finally, the paper proposes policy recommendations that can mitigate risk, ensuring Ethereum's long-term security while preserving the benefits of restaking on capital efficiency.

[4] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

## II. How EigenLayer's Framework Operates

EigenLayer is an innovative middleware protocol that connects Ethereum to decentralized applications (DApps). Its restaking and shared security framework is built on a layered architecture comprising Ethereum validators, Actively Validated Services (AVSs) that leverage validator security through EigenLayer, and DApps that benefit from the services provided by AVSs. This structure presents a significant opportunity to improve capital efficiency while introducing an additional layer of security not available in traditional Proof-of-Stake (PoS) or liquid staking models.

Actively Validated Services (AVSs)

EigenLayer uses AVSs to describe external blockchain services that validators can secure using restaked ETH. AVSs are decentralized applications or protocols, such as oracles (services providing real-world data to blockchains), rollups (speed up transaction processing on the blockchain), cross-chain bridges (connecting different blockchains), and data storage networks that require reliable security but often face high costs and operational challenges in building independent validator networks.

Through EigenLayer, AVSs can easily access Ethereum's established validator base. Instead of constructing separate security frameworks, AVSs utilize the security guarantees provided by Ethereum's validators. This dramatically reduces costs, simplifies setup, and improves reliability, as AVSs leverage an already-trusted network of validators.

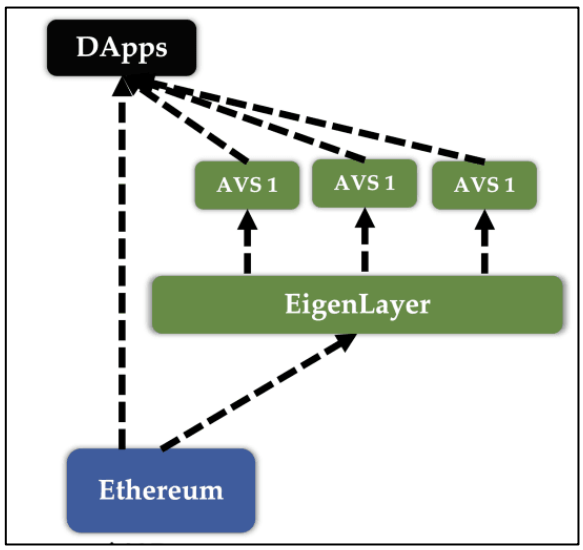
**Example:** EigenDA Data Availability Service, the first AVS launched on EigenLayer, provides a concrete example of how restaking creates shared security. Data availability services ensure blockchain data remains accessible and verifiable, which is crucial for Layer 2 rollups and other scaling solutions. Unlike the current model where each rollup must maintain its own data availability solution, EigenDA leverages Ethereum's validator set through EigenLayer. Validators who opt into EigenDA:

1. Continue their normal Ethereum validation duties
2. Run additional EigenDA software that handles data storage and retrieval requests
3. Make their staked ETH subject to slashing conditions related to data availability duties
4. Earn additional rewards for providing this specialized service

This real-world implementation demonstrates how EigenLayer enables specialized blockchain services without requiring separate validator networks and capital bases.[5]

[5] DAIC Capital. (2025, February 3). *EigenLayer restaking protocol overview*. https://daic.capital/blog/eigen-layer-restaking-protocol

Exhibit 1: EigenLayer Framework[6]




**DApps:** Decentralized applications that interact with the AVSs to retrieve data, execute smart contracts, and maintain network integrity.

**AVSs:** Different services such as oracles, bridges, and rollups that leverage Ethereum's security via EigenLayer.

**EigenLayer:** The middleware layer that enables restaking, allowing validators to extend security to AVSs

**Ethereum:** The base layer of security, where ETH is originally staked.

Restaking and Validator Participation

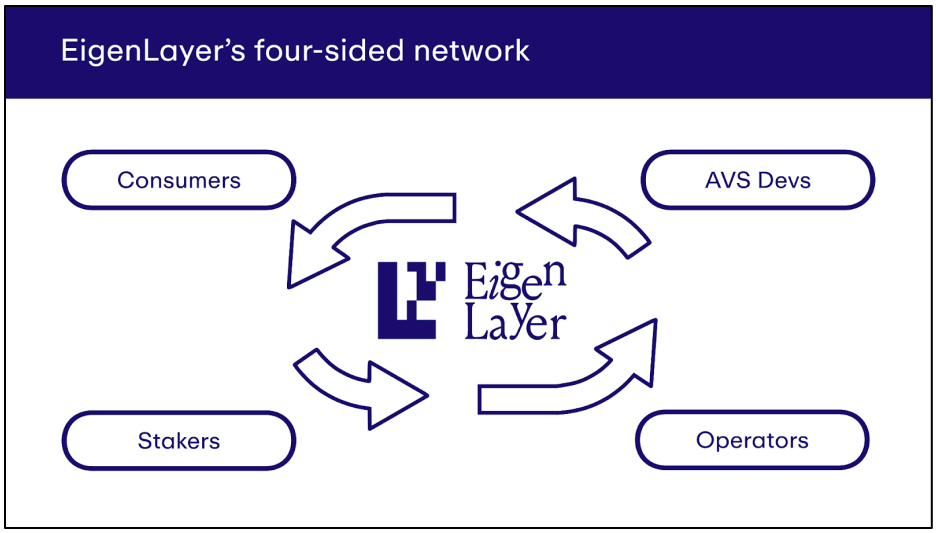
EigenLayer introduces a unique process called restaking, allowing Ethereum validators to reuse previously staked ETH as collateral to secure multiple blockchain services simultaneously. Under Ethereum's current Proof-of-Stake (PoS) model, validators lock up 32 ETH (worth over $50,000 as of early 2025) to validate transactions and maintain network consensus, earning staking rewards in return. This model ties validator capital to Ethereum alone, limiting both earning potential and the utility of staked ETH across broader ecosystems.

Within the EigenLayer ecosystem, off-chain operators perform the actual work of securing external services known as AVSs. ETH stakers on Ethereum's base layer delegate restaked collateral to these operators, thereby extending Ethereum's security guarantees to external protocols. In doing so, operators assume both a portion of the slashing risk and a share of the rewards.

[6] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

The exhibit below illustrates EigenLayer's four-sided network, showing the coordination between stakers, operators, AVS developers, and consumers. At the center, EigenLayer connects these participants through its restaking infrastructure, enabling shared security and decentralized service across the ecosystem.

Exhibit 2: EigenLayer's Four-Sided Network[7]



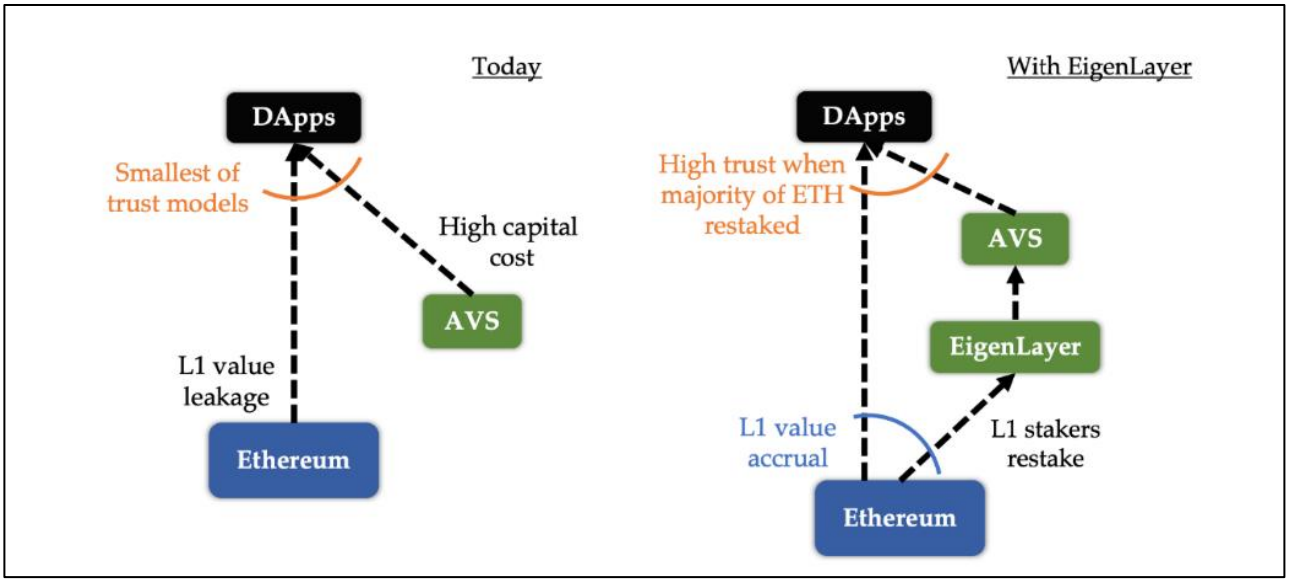

Restaking allows validators to increase their earnings by receiving rewards not only from Ethereum's base protocol, but also from multiple external services simultaneously. For example, a validator staking through EigenLayer has the opportunity to earn Ethereum's base staking APY between 3-5%, plus an additional 0.5-2% per external blockchain service secured, potentially doubling total yield under favorable conditions. To attract validators, these services must offer competitive rewards.

To encourage early use of the platform while AVS rewards remain low, EigenLayer has launched its own governance token, EIGEN. As of early 2025, approximately 4% of the token supply is being distributed through a rewards system. Restakers and node operators receive tokens based on how much ETH they stake and how long they keep it in the system. While EIGEN cannot be traded at launch and is not used for slashing penalties, it is expected to become important in voting on protocol decisions and helping resolve disputes.[8]

[7] EigenLayer. (2025). *EigenLayer Documentation*. https://docs.eigenlayer.xyz/

[8] EigenLayer. (2024). *EigenLayer whitepaper: The Restaking Collective*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

Exhibit 3: EigenLayer's Shared Security Model[9]




Shared Security Model

EigenLayer implements a shared security model, meaning multiple protocols simultaneously rely on Ethereum's validator network rather than creating independent security structures, significantly lowering entry costs for decentralized services while enhancing Ethereum's overall security. Validators opting into EigenLayer agree to additional conditions called "slashing," financial penalties triggered by malicious behavior which ensure their honest participation. If a validator misbehaves, for instance providing false information to an oracle, that validator faces slashing and loses a portion of their staked ETH. By expanding slashing penalties to include behaviors relevant to AVSs, EigenLayer significantly increases the security of multiple protocols.

The shared security model solves a key challenge in blockchain development by allowing protocols to establish trust without building their own validator networks. New blockchain protocols typically struggle to attract enough validators to ensure their security. EigenLayer solves this by allowing new protocols to tap into Ethereum's established security from day one, dramatically lowering the barriers to entry for innovative blockchain services.

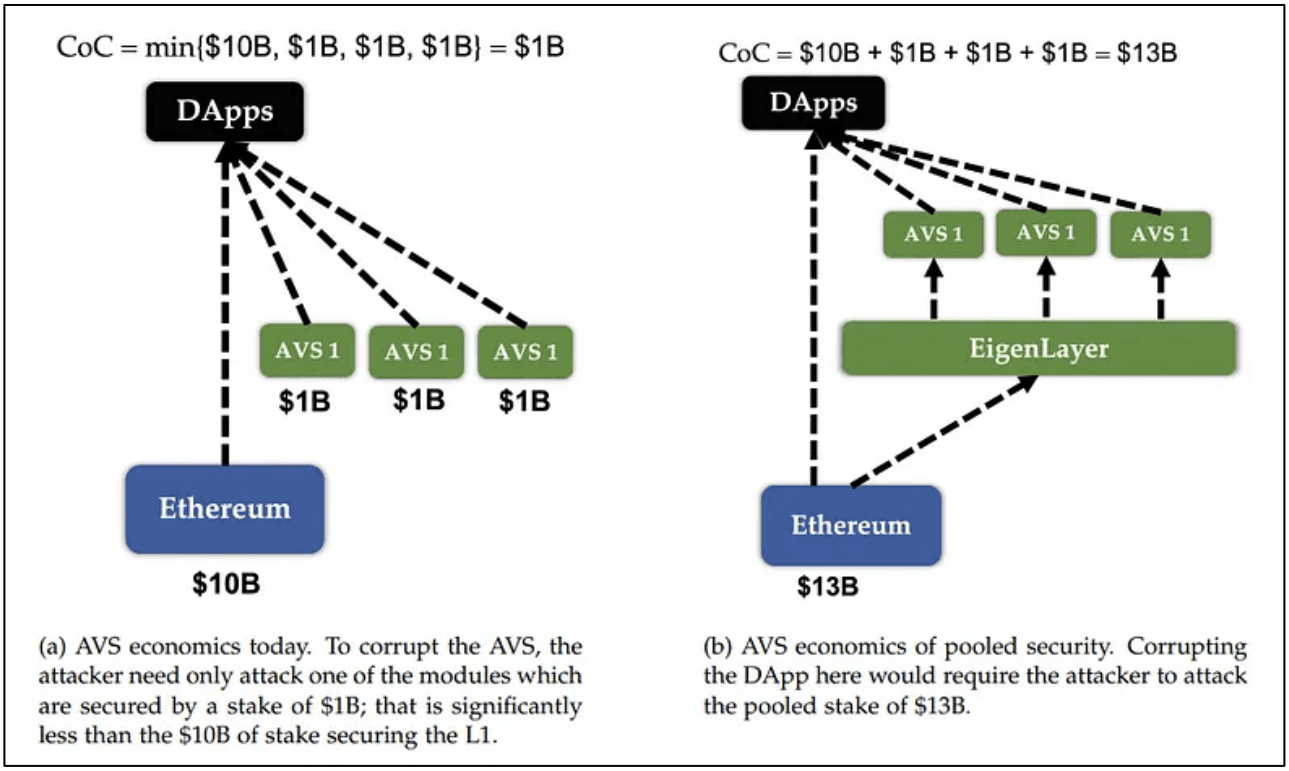
Additionally, this model creates a "cost of corruption" that significantly enhances security. In older validation networks, a validator might be willing to act dishonestly if the potential profit exceeds the penalty. However, with EigenLayer's restaking, the same validator would risk their entire stake across multiple protocols, substantially increasing the cost of corruption and making dishonest behavior an unprofitable strategy in most scenarios.

[9] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective.* https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

Exhibit 4 illustrates the economic security transformation achieved through EigenLayer's pooled security model. The left side (a) shows the current fragmented security landscape, where each AVS maintains its own independent security with minimal capital ($1B each). In this scenario, an attacker needs only to overcome the security of a single AVS ($1B) to compromise the system, even though Ethereum itself has much stronger security ($10B). This represents a significant vulnerability, as the Cost of Corruption (CoC) equals only the minimum security level across all services.

The right side (b) demonstrates EigenLayer's pooled security model, where the same security capital ($13B total) is organized more efficiently. Through restaking, Ethereum validators extend their security to multiple AVSs simultaneously. This pooling creates a shared security layer that an attacker would need to overcome entirely ($13B) to successfully corrupt any individual DApp. This dramatically increases the Cost of Corruption by forcing attackers to overcome the combined security of all participating systems rather than targeting the weakest link. This transformation represents one of EigenLayer's most significant innovations: enabling individual protocols to benefit from the collective security of Ethereum's entire validator set.

Exhibit 4: Pooled Security of EigenLayer[10]




(a) AVS economics today. To corrupt the AVS, the attacker need only attack one of the modules which are secured by a stake of $1B; that is significantly less than the $10B of stake securing the L1.

(b) AVS economics of pooled security. Corrupting the DApp here would require the attacker to attack the pooled stake of $13B.

[10] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

Restaking Risks and Governance Challenges

Additional risks and penalties come with restaking, particularly in slashing and governance:

1. **Cascading Slashing**
   This penalty occurs when a validator's misbehavior in one service triggers a chain reaction of slashing events across multiple AVSs where the same ETH has been restaked. Because the asset is reused as collateral across several services, a single misstep can lead to widespread penalties. For example, if a validator restakes into three AVSs, a 5% penalty could become a 15% loss. If slashed below Ethereum's 32 ETH threshold, validators are forced to exit, losing rewards and weakening the network.[11]

2. **Dual Slashing**
   This situation occurs when validators are penalized by both Ethereum and other AVSs for the same misbehavior. Validators operating within EigenLayer are subject to two distinct rule sets: Ethereum's consensus rules and the slashing conditions defined by each AVS. As a result, the cumulative risk of penalty increases. A single error (such as going offline or submitting invalid data) could result in multiple penalties from both layers. Like cascading slashing, dual slashing compounds financial losses, and could force validators out of the network, creating instability.

3. **Staking Concentration**
   EigenLayer's incentive structure can lead to staking concentration, where validators flock to a small number of high-reward AVSs in hopes of greater financial returns. Concentration increases systemic risk because if a dominant AVS fails, many validators may be hurt at once.

4. **Governance Risk: Complexity and Centralization**
   EigenLayer's restaking model introduces complex governance and potential centralization where many AVSs depend on the same group of validators, each with unique slashing rules and operational requirements. Larger, more powerful AVSs may exert greater influence, shaping policies at the expense of smaller participants. This dynamic could weaken Ethereum by placing control over network security in the hands of a few, undermining the principles of equal access and decentralized participation that define DeFi.

EigenLayer's restaking model boosts capital efficiency but introduces key risks. Validators face slashing, where a single failure can lead to multiple penalties and forced exits. Staking concentration and governance centralization can further weaken the system. The inherent risk of overextended collateralization echoes the risks seen in traditional finance.[12]

[11] ConsenSys. (2024, February 7). *Understanding slashing in Ethereum staking: Its importance and consequences*. https://consensys.io/blog/understanding-slashing-in-ethereum-staking-its-importance-and-consequences

[12] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective.* https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

## III. Comparing EigenLayer to Traditional Staking Models and Financial Practices

From Traditional Staking to Liquid and Restaked Capital

As introduced in earlier sections, Ethereum's Proof-of-Stake (PoS) system requires validators to lock 32 ETH, earning modest yields while limiting capital flexibility. This section builds on that foundation by comparing EigenLayer's restaking model to other staking paradigms, specifically liquid staking platforms such as Lido and Rocket Pool, custodial exchanges, and newer restaking protocols. It also traces the evolution from Liquid Staking Tokens (LSTs) to Liquid Restaking Tokens (LRTs), examining how these tools expand capital utility while introducing new layers of systemic risk.

To address this, platforms such as Lido and Rocket Pool pioneered liquid staking, which issues Liquid Staking Tokens (LSTs) such as stETH or rETH to users in exchange for staked ETH. These tokens can be traded or deployed in DeFi while still accruing staking rewards. LSTs solved the problem of illiquidity, but only partially. Even when used in lending or farming protocols, this capital continues to secure only the Ethereum base layer.

EigenLayer builds on this model through restaking. Rather than merely holding stETH, users can now "restake" it to help secure new Actively Validated Services (AVSs), These include oracles, bridges, rollups, and other infrastructure protocols that traditionally needed to build their own validator sets. EigenLayer enables Ethereum's validator set to become a multi-service staking layer, improving capital efficiency by reusing the same collateral across multiple services.

Newer platforms such as Ether.Fi, Renzo, and Kelp DAO extend this system even further with Liquid Restaking Tokens (LRTs), which represent ETH that has already been restaked in EigenLayer. These LRTs reintroduce liquidity to restaked capital, allowing it to be reused once more in DeFi, creating a powerful but potentially fragile stack of yield-generating layers.[13]

[13] CoinMarketCap. (2023). *The ultimate guide to Ethereum liquid staking*. https://coinmarketcap.com/academy/article/the-ultimate-guide-to-ethereum-liquid-staking

Exhibit 5: Platform Comparisons - EigenLayer vs. Lido, Rocket Pool, and Centralized Services [14]

| Platform | Model | Rewards (APY) | Capital Efficiency | Decentralization | Key Risks |
|---|---|---|---|---|---|
| Lido | Liquid Staking Protocol | ~3.06% | Medium–High | Medium | Centralized operators, smart contract risk |
| Rocket Pool | Decentralized Liquid Staking | ~2.83% | Medium–High | High | Node diversity, protocol complexity |
| Coinbase/Kraken | Centralized Exchange | 2.15–7% | Low | Low | Custodial risk, regulatory exposure |
| EigenLayer | Restaking Middleware | ~5% (est.) | High | Medium–High | Cascading slashing, governance centralization |

Lido and Rocket Pool allow ETH holders to earn staking rewards while maintaining liquidity. However, these platforms do not expand the role of staked ETH beyond Ethereum itself. EigenLayer introduces a new layer of utility: multi-service security. Validators and delegators can restake ETH or stETH to earn additional rewards from AVSs. This opportunity, however, comes with the risk of additional slashing risk proportional to the number and type of AVSs secured.

Why Not Use stETH in DeFi Instead?

Many ETH holders choose to deploy stETH in DeFi protocols such as Aave or Curve, where it can generate yield through lending or providing liquidity. These methods provide flexibility and short-term financial returns but do not enhance Ethereum's network security. In contrast, EigenLayer enables capital to simultaneously earn yield and strengthen essential infrastructure such as oracles and rollups, aligning incentives between economic activity and the broader utility of the blockchain network.

[14] *Sources: Lido, Rocket Pool, Coinbase, Kraken, EigenLayer documentation; Kiln, Cointelegraph, ARPA, EtherFi data (as of April 2025).*

That said, this yield is not "free." Restakers face AVS-specific slashing conditions, governance complexity, and longer withdrawal timelines. The table below outlines the capital trade-offs:

Exhibit 6: Where to Deploy stETH - Comparing Returns, Risks, and Ecosystem Impact [15]

| **Strategy** | **Yield Potential** | **Liquidity** | **Capital Efficiency** | **Systemic Risk** | **Ethereum Security Contribution** |
|---|---|---|---|---|---|
| stETH → Aave/Curve | Moderate | High | Medium | Moderate | None (beyond base consensus) |
| stETH → EigenLayer | High | Low–Medium | High | High | Yes |
| stETH → EigenLayer → LRT → DeFi | Very High | High | Very High | Very High | Yes (with exposure) |

LRTs: Stacking Yield, Stacking Risk

The emergence of Liquid Restaking Tokens (LRTs) adds a third layer of flexibility and integration. Users can now take restaked ETH and receive a tokenized representation, e.g., eETH (EtherFi), ezETH (Renzo), or rsETH (Kelp DAO), that can be deployed in additional DeFi protocols. This allows yield stacking across:

- Ethereum base staking rewards
- AVS protocol incentives (in ETH or native tokens)
- EIGEN incentives
- DeFi lending/farming returns

However, this layered structure also concentrates exposure. LRT holders are indirectly exposed to:

- Slashing risk from multiple AVSs
- Governance or oracle failure
- Price volatility of the LRT itself
- Smart contract failure or redemption risk

LRTs introduce a form of collateral chaining that resembles the structured finance models from before the 2008 financial crisis that were financially productive, but systemically fragile.

[15] *Sources: Based on data from Kiln.fi, EigenLayer docs, Cointelegraph, Lido, Rocket Pool, DeFiLlama & public LRT documentation as of April 2025.*

Traditional Finance Parallel: Rehypothecation and Systemic Fragility

EigenLayer's restaking model mirrors the financial practice of rehypothecation, where the same collateral is pledged multiple times. While this can unlock capital efficiency, it also introduces the risk of cascading failures across the system. In the 2008 financial crisis, institutions like Lehman Brothers reused the same mortgage-backed securities across multiple obligations. When the value of subprime mortgage assets collapsed, these overleveraged positions triggered widespread defaults, liquidity freezes, and systemic panic.[16]

EigenLayer could face similar vulnerabilities. If multiple AVSs fail or act maliciously, validators may incur cascading slashing penalties that affect associated delegators and LRT holders. Without safeguards, such as exposure caps, AVS segregation, and adaptive slashing, this model could destabilize Ethereum's validator set and erode trust in the network.[17]

[16] Investopedia. (2023, January 26). *Rehypothecation*. https://www.investopedia.com/terms/r/rehypothecation.asp

[17] Wikipedia contributors. (n.d.). *Financial crisis of 2007–2008*. Wikipedia. https://en.wikipedia.org/wiki/Financial_crisis_of_2007%E2%80%932008

## IV. EigenLayer Simulation Analysis: Quantifying Systemic Risk in Restaking

To quantify the potential impact of EigenLayer's restaking model on Ethereum's security, a Monte Carlo simulation was developed modeling 1,000 validators across multiple failure scenarios. This section presents the methodology, key findings on validator behavior, and implications for systemic risk management in EigenLayer's ecosystem.

Simulation Framework and Design

The simulation was designed to stress-test EigenLayer's shared security model by examining how validator returns, and network stability respond to varying levels of AVS failures. Using a comprehensive Monte Carlo approach with 1,000 iterations, a realistic model of the EigenLayer ecosystem was created with the following parameters:

- **Validator Population:** 1,000 unique validators with varying risk profiles
- **Available AVSs:** 100 potential services with varied reward and risk profiles
- **Base Ethereum APR:** 3.0% (baseline staking rewards)
- **AVS Reward Range:** 0.25-1.0% additional APR per AVS
- **Correlation Factor:** 15% (increasing penalty multiplier for each additional failed AVS)
- **Simulation Period:** One full year of operation

Validators were distributed across four risk categories to reflect realistic market behavior:

1. **Conservative (40% of validators):** Participating in only 1-5% of available AVSs
2. **Moderate (35% of validators):** Balancing risk/reward with 3-10% AVS participation
3. **Aggressive (20% of validators):** Pursuing higher yields via 5-25% AVS participation
4. **Ultra-aggressive (5% of validators):** Maximizing yield through 10-100% AVS participation

To test system resilience, four progressively severe failure scenarios were modeled:

1. **No AVS failure (Base Case):** Ideal conditions with perfect AVS performance
2. **Single AVS failure (Isolated Incident):** One random AVS experiences failures
3. **Five AVS failures (Correlated Slashing):** Multiple simultaneous failures affecting a significant portion of the network
4. **Twenty AVS failures (Extreme Network Event):** Catastrophic, system-wide failure

For each scenario, comprehensive metrics were tracked including operator APR distribution, slashing probability, and system-wide economic impact.[18]

[18] Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

Risk-Return Analysis by Validator Profile

The simulation reveals striking differences in risk-return outcomes across validator profiles and failure scenarios, illustrating EigenLayer's fundamental risk-reward trade-off, which underscores the need for the safeguards discussed in this paper.

Base Case: Capital Efficiency Benefits

In the no-failure scenario, restaking dramatically enhances validator returns without corresponding risk exposure. Conservative validators earn approximately 4-5% APR (a 1-2 percentage point improvement over standard Ethereum staking), while ultra-aggressive validators achieve returns of 20-40% APR by participating in dozens of AVSs. This scenario represents the core value proposition of EigenLayer: validators can multiply their earnings by utilizing the same staked ETH to secure multiple protocols. The simulation confirms that in normal operating conditions, EigenLayer significantly improves capital efficiency, validating a key element of this paper.

Single Failure: Early Warning Signs

The introduction of even a single AVS failure dramatically alters the risk-return profile. While the median returns remain positive across all validator profiles, the distribution widens considerably for aggressive and ultra-aggressive validators. The data shows that 54.7% of ultra-aggressive validators experienced slashing in this scenario, compared to just 2.6% of conservative validators. This stark difference highlights how quickly the "free yield" narrative breaks down once risk materializes. The single failure scenario serves as an early warning that aggressive restaking strategies carry substantial hidden risk, even under relatively mild stress conditions.

Cascade Scenario: The Importance of Adaptive Slashing Mechanisms

The five-failure scenario represents a critical inflection point that emphasizes the importance of adaptive slashing mechanisms, one of the key safeguards discussed in this paper. The simulation shows that nearly a quarter (24.8%) of all validators face slashing, with the percentage much higher among aggressive and ultra-aggressive validators.

Exhibit 7: Slashing Probability by Validator Risk Profile (Cascade Scenario) [19]

| **Risk Profile** | **% Slashed (Cascade Scenario)** |
|---|---|
| Conservative | 12.0% |
| Moderate | 26.9% |
| Aggressive | 52.8% |
| Ultra-aggressive | 90.0% |

[19] Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

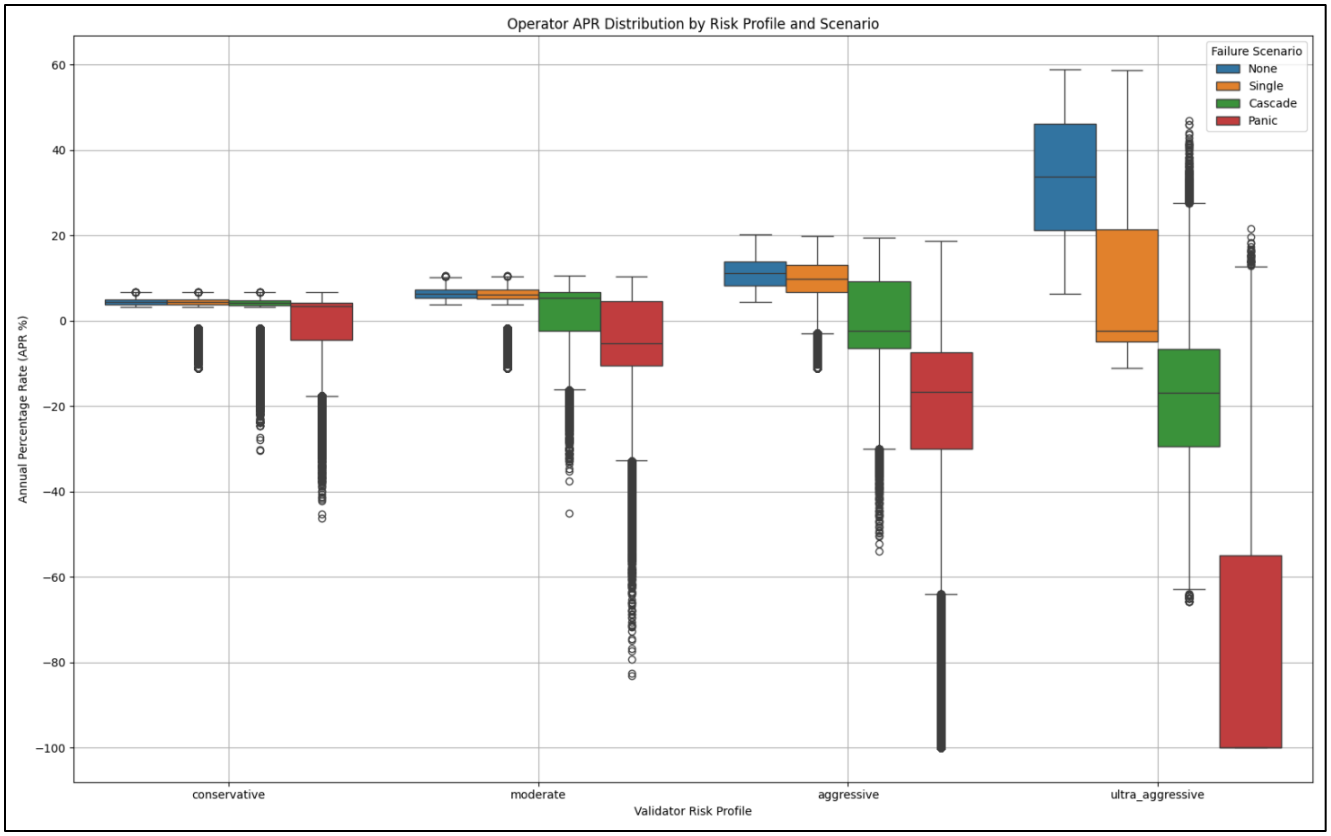
These results indicate that validators need to carefully manage their AVS exposure. With the implementation of features like "Unique Stake Allocation" that EigenLayer is developing, validators could isolate their risk across different AVSs, preventing the cascade effect and preserving the capital efficiency benefits of restaking even during periods of moderate market stress.

Panic Scenario: Systemic Collapse

Under catastrophic conditions with 20 AVS failures, the simulation shows a near-complete reversal of the base case dynamics. Even conservative validators face substantial slashing risk (41.3%), with negative median returns across almost all validator profiles. The most striking result is that 99.6% of ultra-aggressive validators experience slashing, with catastrophic losses approaching -100% APR in extreme cases. This represents a complete wipeout of validator stake, likely triggering forced exits from Ethereum's validator set.

This extreme scenario, while unlikely, highlights the importance of the economic safeguards discussed in this paper, such as staking caps and liquidity buffers. With these protections, the impact of even a severe market disruption could be contained, preserving Ethereum's security while maintaining the capital efficiency of restaking under normal conditions.

Exhibit 8: APR Outcomes by Scenario and Validator Type[20]



Correlation Effects and Cascading Slashing

A critical finding from the simulation is the non-linear amplification of slashing penalties when validators face multiple simultaneous failures. The correlation factor implemented in the model at 15% demonstrates how penalties compound exponentially as validators encounter multiple AVS failures. The relationship between number of failed AVSs and slashing penalties is strongly non-linear. While a single AVS failure might result in a modest 5-7% loss, validators experiencing 5-10 failed AVSs face exponentially greater penalties approaching the maximum possible loss. This correlation effect creates a dangerous dynamic similar to rehypothecation in traditional finance where, rather than providing diversification, exposure to multiple AVSs amplifies risk during systemic events. The data shows that under correlated failures, the average slashing penalty for affected validators was 24.7% of stake, far exceeding what would be expected from independent events.

Optimal AVS Participation Thresholds

To provide practical guidance for validators, the relationship between AVS participation count and slashing probability was analyzed across different risk tolerance levels. This analysis reveals clear thresholds for safe participation in EigenLayer's ecosystem. The probability of experiencing at least one slashing event grows non-linearly with the number of AVSs a validator secures. For each risk tolerance level, maximum safe participation thresholds were identified:

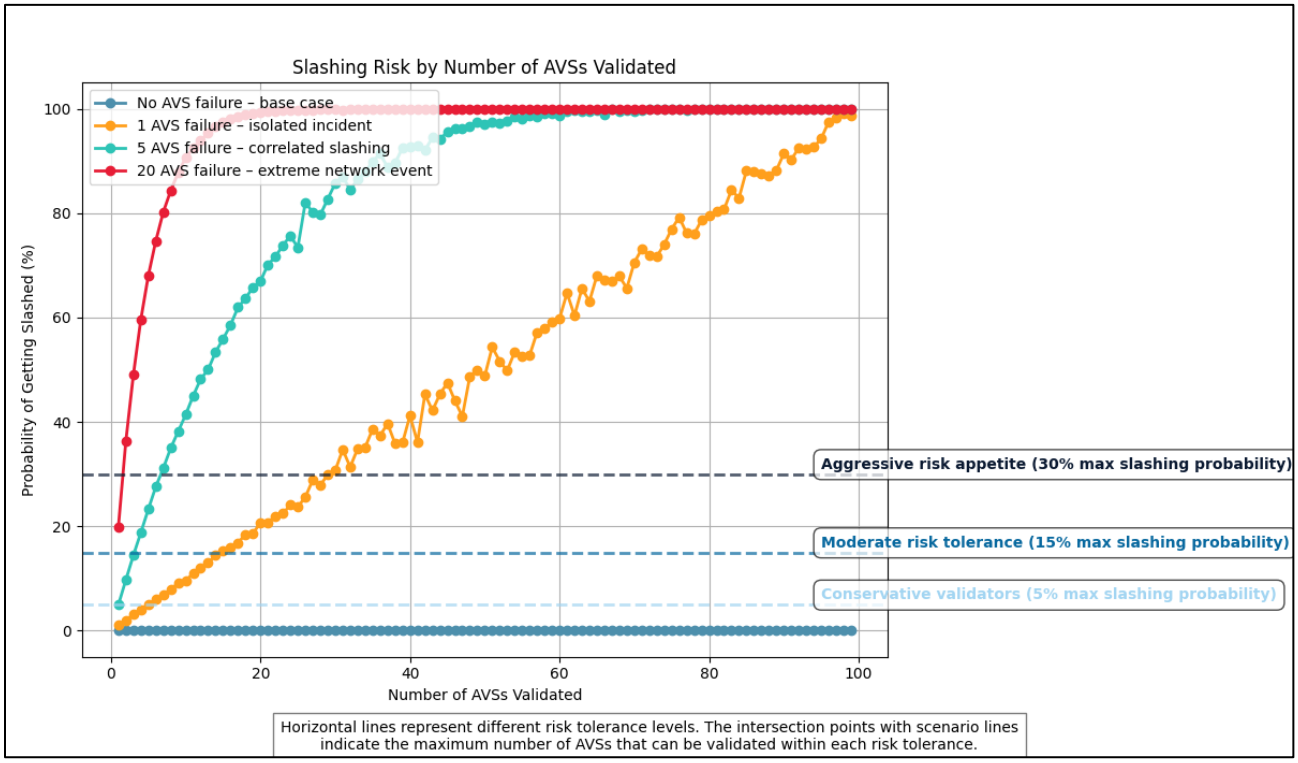
Exhibit 9: Maximum Safe Participation Thresholds[21]

| Risk Tolerance Level | Single Failure Scenario | Cascade Scenario | Panic Scenario |
|---|---|---|---|
| Conservative (≤5%) | ≤8 AVSs | ≤2 AVSs | <1 AVS |
| Moderate (≤15%) | ≤24 AVSs | ≤5 AVSs | ≤2 AVSs |
| Aggressive (≤30%) | ≤48 AVSs | ≤11 AVSs | ≤3 AVSs |

Note: The risk appetite categories used here (≤5%, ≤15%, ≤30%) are slashing probability thresholds and not directly equivalent to the behavioral risk profiles defined earlier.

[20 & 21] Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

Exhibit 10: Slashing Risk by Number of AVSs Validated[22]




These thresholds indicate that validators should carefully limit their AVS exposure based on their risk tolerance and potential failure scenarios. Even validators pursuing aggressive restaking strategies should not exceed participation in more than 11 AVSs when accounting for potential systemic failures. Adhering to these guidelines enables a more optimized risk-reward profile while maintaining contributions to the overall security of the Ethereum ecosystem.

Systemic Impact on Ethereum Security

Beyond individual validator outcomes, the simulation allows for projecting system-wide impacts on Ethereum's security. By aggregating individual validator results, it was possible to estimate how different failure scenarios might affect the overall staked ETH securing Ethereum's consensus.

[22] Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

Exhibit 11: Validator Slashing and Projected ETH Network Stake Reduction [23]

| Scenario | % of Validators Slashed | Avg. Stake Loss Per Slashed Validator | Projected ETH Network Stake Reduction |
|---|---|---|---|
| No Failure | 0.0% | 0.0% | 0.0% |
| Single Failure | 8.8% | 6.5% | 0.6% |
| Cascade (5 Failures) | 29.3% | 24.7% | 7.2% |
| Panic (20 Failures) | 65.2% | 41.2% | 26.9% |

These projections highlight a concerning vulnerability: in the cascade scenario, approximately 7.2% of the total staked ETH could be at risk, while the panic scenario could theoretically reduce Ethereum's economic security by over a quarter. This represents a novel form of systemic risk that did not exist in Ethereum's security model before the introduction of restaking.

The distribution of slashing events across validator profiles raises additional concerns about network centralization. If a correlated slashing event disproportionately affects smaller, independent validators (who may be more likely to pursue aggressive restaking strategies for higher yields), it could lead to a more concentrated validator set dominated by conservative institutional stakeholders who avoided high exposure restaking.

Simulation Insights Supporting Thesis

The simulation results offer strong support for this paper's thesis by demonstrating that:

1. **Significant Capital Efficiency Improvements**
   In the base case scenario, EigenLayer dramatically improves returns for validators (up to 90% improvement for aggressive strategies) without introducing additional risk, confirming its ability to enhance capital efficiency.

2. **Manageable Risks with Proper Safeguards**
   Even in stress scenarios, validators employing conservative strategies and following recommended AVS participation limits maintain positive risk-adjusted returns, indicating that risks can be effectively managed.

3. **Economic Benefits Outweigh Potential Risks**
   The simulation shows that with appropriate risk management, the economic benefits of restaking outweigh the potential risks for most validators across most scenarios.

[23] Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

4. **Effectiveness of Proposed Safeguards**
   The data strongly supports the effectiveness of the safeguards proposed in this paper, particularly adaptive slashing mechanisms, decentralized governance frameworks, and economic safeguards, in mitigating the identified risks.

5. **Preservation of Ethereum's Security**
   While extreme scenarios could potentially impact Ethereum's security, the likelihood and severity of such impacts can be significantly reduced through the implementation of the recommended safeguards.

These findings confirm that, while EigenLayer introduces systemic risks similar to those in traditional finance, its commitment to integrating necessary safeguards will allow it to significantly improve crypto-economic capital efficiency without fundamentally undermining Ethereum's security.

Limitations and Assumptions

While the simulation provides valuable insights into the risks associated with EigenLayer, several important limitations and assumptions should be acknowledged:

1. **Simplified Failure Probabilities**
   The simulation uses random failure selection rather than modeling specific failure probabilities for each AVS. In reality, different AVSs would have varying security standards and failure risks.

2. **Homogeneous AVS Risk Treatment**
   The model treats all AVSs as having similar risk profiles, whereas in reality, different AVSs would have varying technical complexity and failure probabilities.

3. **Simplified Validator Behavior**
   The simulation assigns static risk profiles to each validator, assuming that behavior remains consistent throughout the simulation period. In real-world conditions, validator strategies would likely evolve dynamically in response to network incentives and observed AVS performance.

4. **Linear Correlation Model**
   The correlation factor implementation (15% increase in penalty per additional failed AVS) is a simplified approach to modeling complex interdependencies. Real-world correlations might follow more complex patterns.[24]

[24] Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

5. **Perfect Information Assumption**
   The model assumes validators have perfect information about risk-return trade-offs when selecting AVSs. In reality, faulty or incomplete information would influence validator decision-making.

While the simulation provides valuable directional insights, its specific numerical predictions should be interpreted with these limitations in mind. Further research with more complex, AVS-specific risk modeling and dynamic validator behavior would further enhance understanding of EigenLayer's long-term systemic implications.

The simulation quantifies the double-edged nature of EigenLayer's restaking model. While EigenLayer offers significant capital efficiency improvements and yield enhancement in ideal conditions, it introduces complex systemic risks that must be managed through appropriate safeguards. The data strongly suggests that with the implementation of adaptive slashing mechanisms, decentralized governance frameworks, and economic safeguards, EigenLayer can achieve its goal of expanding capital efficiency without undermining Ethereum's fundamental security.[25]

[25] Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

## V. EigenLayer's Safeguards Against Systemic Risks

EigenLayer's restaking model offers promising capital efficiency gains but introduces significant systemic risks, including cascading slashing, validator overexposure, and governance complexity. To evaluate whether these risks fundamentally threaten Ethereum's security, or can be effectively managed, this section combines a review of EigenLayer's current safeguards with original proposals for protocol-level improvements. Taken together, these mechanisms constitute a comprehensive risk management framework aimed at ensuring EigenLayer's long-term viability and resilience.

Addressing Slashing and Validator Overexposure

To mitigate the risks of slashing and validator overexposure, EigenLayer uses several protective strategies. Validators can allocate specific portions of their stake to individual Actively Validated Services (AVSs), so a slashing event in one AVS affects only the portion assigned to it rather than the validator's entire stake. EigenLayer also pools over $7 billion in restaked ETH, which increases the cost of attacking the network and makes malicious behavior economically unfavorable.[26] AVSs can acquire slashing redistribution rights that allow funds to be returned to users in cases of honest validator mistakes, turning slashing from a fully punitive penalty into one that can be partially recovered. For example, if a bridge service acquires $25 million in redistribution rights and a validator it depends on is slashed, those funds could be used to offset user losses. In addition, tools like Puffer's anti-slasher libraries help validators avoid common technical errors that lead to slashing, lowering barriers to entry and supporting broader, more decentralized participation.[27]

Countering Centralization Risks

To prevent centralization and promote equitable participation, EigenLayer has implemented several measures. It has established the Decentralized Governance Council (EigenGov), a group of domain experts chosen by EIGEN token holders to make protocol decisions, ensuring decision-making remains decentralized. By promoting AVSs with minimal hardware requirements, such as EigenDA, EigenLayer lowers technical barriers and supports participation from smaller operators. EigenLayer also offers reward incentives and lower minimum stake requirements for solo validators, who help strengthen the network and reduce the risk of failures caused by too few validators.[28]

[26] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective.* https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

[27] ConsenSys. (2024, February 7). *Understanding slashing in Ethereum staking: Its importance and consequences*. https://consensys.io/blog/understanding-slashing-in-ethereum-staking-its-importance-and-consequences

[28] Galaxy Digital. (2025). *Restaking risks and rewards*. https://www.galaxy.com/research/

Managing Validator Integrity and Conflicts of Interest

To prevent validator misconduct and conflicts of interest, EigenLayer employs several key tools. Trusted Execution Environments (TEEs) and shared anti-slasher libraries help validators follow the rules of each AVS, limiting the potential for misconduct. Validators are also required to post collateral, aligning their financial interests with the overall health of the protocol. To limit risk, EigenLayer sets staking caps and requires liquidity buffers, both of which help reduce exposure and mitigate the impact of malicious behavior. Real-time dashboards, third-party audits, and open discussions concerning governance facilitate transparency and community oversight, allowing stakeholders to monitor validator behavior and respond quickly to potential risks.

The Role of EIGEN in Decision-Making

The EIGEN token serves a pivotal role in governance and decision-making. EIGEN holders can vote on critical protocol policies, including which AVSs are approved and how to handle disputes, ensuring that important decisions reflect community consensus. EIGEN also allows participants to respond to issues that cannot be solved automatically with code, such as problems with data feeds or disagreements between services. In serious cases, the community can work together to take actions like creating a new version of the token to punish dishonest behavior. Together, ETH and EIGEN support EigenLayer's security model: ETH serves as the financial stake that keeps validators honest, while EIGEN helps the community manage governance and resolve problems when needed.[29]

Proposals for Strengthening Safeguards

While EigenLayer's current safeguards offer a strong foundation, there are several enhancements that could further reduce systemic risk and improve validator confidence. AVSs could be assigned adaptive risk scores based on metrics such as uptime, slashing history, and dispute frequency, which would be made public through a transparent dashboard to help validators compare risk-adjusted returns and tailor their exposure to their risk tolerance. A decentralized slashing insurance fund, governed by EIGEN holders and funded by risk-based premiums, could offer compensation to honest validators affected by large-scale failures. Introducing a tiered slashing system would allow the protocol to apply lighter penalties for minor errors and stronger consequences for repeated or malicious behavior, helping retain honest participants without over-penalizing accidental behavior. Finally, a live governance dashboard displaying AVS onboarding, slashing appeals, insurance fund proposals, and council activity would improve transparency, encourage community involvement, and strengthen trust in decision-making. Together, these improvements support the paper's central thesis: EigenLayer can enhance capital efficiency while preserving Ethereum's security.

[29] EigenLayer. (2023). *EigenLayer whitepaper: The Restaking Collective*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

## VI. Conclusion: Balancing Innovation and Risk

EigenLayer represents an ambitious and innovative development in Ethereum's ecosystem. By allowing restaking, a mechanism that allows validators to extend their staked ETH to secure multiple decentralized services, it significantly improves capital efficiency and unlocks new revenue opportunities for validators, attracting them to the network.

However, this innovation introduces notable risks. Reusing ETH across multiple services mirrors risky collateral practices seen in margin trading and rehypothecation. Validators who restake too broadly face heightened exposure, increasing the likelihood of cascading slashing where one service failure triggers widespread penalties. This dynamic, along with collateral overextension, validator centralization, and governance challenges, could erode Ethereum's core values of decentralization and security.

This paper has explored these dynamics through an analysis of EigenLayer's framework and restaking model, comparisons with other network models and traditional financial practices, and a custom simulation of validator behavior. The analysis showed that the restaking model is a more efficient use of capital and provides an extra layer of shared security that other models do not have. The simulation results clearly demonstrate that operators who overextend themselves across too many services face substantial risks during periods of market stress, with cascading penalties that can quickly erase any additional rewards earned through restaking. Yet the results also suggest that with appropriate risk management strategies and moderately conservative approaches to service participation, validators can meaningfully enhance their returns while keeping risks manageable.

EigenLayer's restaking model requires careful balance**.** Safeguards against slashing fallout, governance centralization, and conflicts of interest are essential to network stability. While the model introduces systemic risks similar to those in traditional finance, these can be mitigated through thoughtful, well-implemented protections. With EigenLayer's commitment to such safeguards, restaking offers a meaningful advancement in crypto-economic capital efficiency for Ethereum. By balancing innovation with security, EigenLayer can significantly expand capital efficiency without undermining the network's foundational security.

## Bibliography


Bachini, J. (2023, November 2). EigenLayer. https://jamesbachini.com/eigenlayer/

Beaconcha.in. (2025). *Ethereum validator data explorer*. https://beaconcha.in

Chainlink. (2025). *Chainlink staking v0.1 overview: Securing oracle networks with staked LINK*. https://chain.link/staking

Coinbase. (2025). *Guide to EigenLayer: Restaking explained*. https://www.coinbase.com/learn/crypto-basics/what-is-eigenlayer

CoinMarketCap. (2023). *The ultimate guide to Ethereum liquid staking*. https://coinmarketcap.com/academy/article/the-ultimate-guide-to-ethereum-liquid-staking

Consensys. (2024, May 29). E*igenLayer: Decentralized Ethereum restaking protocol explained.* https://consensys.io/blog/eigenlayer-decentralized-ethereum-restaking-protocol-explained

ConsenSys. (2024, February 7). *Understanding slashing in Ethereum staking: Its importance and consequences.* https://consensys.io/blog/understanding-slashing-in-ethereum-staking-its-importance-and-consequences

DAIC Capital. (2025, February 3). *EigenLayer restaking protocol overview.* https://daic.capital/blog/eigen-layer-restaking-protocol

Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation

EigenLayer. (2023). *EigenLayer whitepaper: The Re-staking Collective*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

EigenLayer. (2025). *EigenLayer Documentation*. https://docs.eigenlayer.xyz/

Ethereum Foundation. (2025). *Staking on Ethereum.* https://ethereum.org/en/staking/

Galaxy Digital. (2025). *Restaking risks and rewards.* https://www.galaxy.com/research/

Gandhi, D. (2023, August 15). *Drop #62: EigenLayer's restaking revolution*. https://darshang.substack.com/p/drop-62-eigen-layer

Investopedia. (2023, January 26). *Rehypothecation*. https://www.investopedia.com/terms/r/rehypothecation.asp

McKinney, J. (2023). *EigenLayer explained: The fourth paradigm in crypto-economic capital efficiency*. https://www.youtube.com/watch?v=iMFscq9Sxdk

Reflexivity Research. (n.d.). *Exploring EigenLayer.* Reflexivity Research. https://www.reflexivityresearch.com/free-reports/exploring-eigenlayer

Rocket Pool. (2025). *Staking via a Decentralized Exchange on the Ethereum Network (Layer 1).* https://docs.rocketpool.net/guides/staking/via-l1.html

TokenInsight. (2024, October 14). *EigenLayer's Eigen Token Analysis.* https://tokeninsight.com/en/research/analysts-pick/eigenlayer-s-eigen-token-analysis

Wikipedia contributors. (n.d.). *Financial crisis of 2007–2008.* Wikipedia. https://en.wikipedia.org/wiki/Financial_crisis_of_2007%E2%80%932008

Note: Certain sections of this research paper were revised with the help of AI writing tools to improve clarity, organization, and readability.

Anthropic. (2025). *Claude 3.7 Sonnet* [Large language model]. https://claude.ai

OpenAI. (2025). *ChatGPT 4o* [Large language model]. https://chat.openai.com

## Appendix: Simulation Summary Statistics and Code

Exhibit 13: Summary Statistics Across Scenarios

| Scenario | Description | % Slashed | Avg Op APR | Med Op APR | Avg Del APR | Std Dev APR | Min APR | Max APR | Q1 APR | Q3 APR | % Slashed (aggr.) | % Slashed (cons.) | % Slashed (mod.) | % Slashed (ultra) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **none** | No failures (base case) | 0.00 | 0.079 | 0.056 | 0.067 | 0.074 | 0.032 | 0.588 | 0.045 | 0.079 | 0.00 | 0.00 | 0.00 | 0.00 |
| **single** | Single AVS failure | 8.77 | 0.058 | 0.052 | 0.049 | 0.055 | -0.111 | 0.588 | 0.041 | 0.072 | 14.52 | 2.57 | 5.99 | 54.71 |
| **cascade** | 5 AVS failures (correlated) | 29.27 | 0.016 | 0.044 | 0.014 | 0.084 | -0.656 | 0.469 | -0.028 | 0.058 | 52.80 | 12.03 | 26.87 | 90.00 |
| **panic** | 20 AVS failures (extreme) | 65.19 | -0.100 | -0.046 | -0.085 | 0.216 | -1.000 | 0.216 | -0.124 | 0.040 | 92.59 | 41.25 | 71.97 | 99.58 |

The following Python code was used to conduct the Monte Carlo simulation of EigenLayer operators under various slashing scenarios. The full code repository is available at: https://github.com/bserickson/eigenlayer_simulation

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
import seaborn as sns
import random
from collections import defaultdict
import os
from tqdm import tqdm
from datetime import datetime
import warnings
import logging
logging.getLogger('matplotlib').setLevel(logging.ERROR)

# ---------------------------------------------
# EigenLayer Simulation
# ---------------------------------------------
# Simulates economic outcomes for EigenLayer operators who restake ETH across AVSs (Actively Validated Services).
# Monte Carlo simulation models economic risks (slashing) and returns (APR) under multiple failure scenarios.
# Outputs include scenario-based CSVs, summary statistics, and high-quality visualizations for analysis/reporting.

# ---------------------------------------------
# SECTION 0: Seed Control for Reproducibility
# ---------------------------------------------
# Allows switching between reproducible results (SEED = 42) or randomized results using current timestamp
USE_RANDOM_SEED = False          # Set to True to use a new seed every time you run, False for fixed seed
SEED = int(datetime.now().timestamp()) if USE_RANDOM_SEED else 42
np.random.seed(SEED)             # Makes NumPy random outputs reproducible
random.seed(SEED)                # Makes Python random outputs reproducible

# ---------------------------------------------
# SECTION 1: Global Parameters and Initialization
# ---------------------------------------------
NUM_OPERATORS = 1000     # Number of simulated operators staking via EigenLayer
NUM_AVS = 100            # Number of Actively Validated Services (AVSs)
ETH_APR = 0.03           # Base Ethereum staking return (APR)
NUM_RUNS = 1000          # Monte Carlo iterations per scenario for robustness
CORRELATION_FACTOR = 0.15  # Increase slashing probability for each additional failed AVS

# ---------------------------------------------
# Print Simulation Overview to Terminal
# ---------------------------------------------
# This provides a readable summary of the simulation parameters before execution,
# useful for understanding the run context, debugging, or replicating results later.
# Includes operator count, AVS count, APR assumption, iteration count, and seed.
print("=" * 60)
print("Running EigenLayer Operator Simulation")
print("- Simulates economic outcomes from restaking ETH across AVSs")
print("- Models operator rewards and slashing risks under 4 scenarios")
print(f"- Monte Carlo simulation with {NUM_OPERATORS} operators × {NUM_RUNS} iterations")
print(f"- {NUM_AVS} Actively Validated Services (AVSs) simulated")
print(f"- Assumes Ethereum base staking APR = {ETH_APR:.2%}")
print(f"- Using correlation factor = {CORRELATION_FACTOR} for recursive slashing")
```

```
print(f"- Simulation SEED: {SEED}")
print("=" * 60)
print("=" * 60)

# ---------------------------------------------
# Continue SECTION 1: Global Parameters and Initialization
# ---------------------------------------------
# Four failure scenarios: increasing in severity
SCENARIOS = ["none", "single", "cascade", "panic"]
SCENARIO_LABELS = {
    "none": "No AVS failure – base case",
    "single": "1 AVS failure – isolated incident",
    "cascade": "5 AVS failure – correlated slashing",
    "panic": "20 AVS failure – extreme network event"
}

# Create timestamped directory to store output
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
results_dir = f"results_{timestamp}"
os.makedirs(results_dir, exist_ok=True)

# Create subfolder for all generated visualizations
visuals_dir = os.path.join(results_dir, "visualizations")
os.makedirs(visuals_dir, exist_ok=True)

# Save configuration parameters for reference and reproducibility
with open(f"{results_dir}/config.txt", "w") as f:
    f.write("=== Simulation Parameters ===\n")
    f.write(f"SEED: {SEED}\n")
    f.write(f"NUM_OPERATORS: {NUM_OPERATORS}\n")
    f.write(f"NUM_AVS: {NUM_AVS}\n")
    f.write(f"ETH_APR: {ETH_APR}\n")
    f.write(f"NUM_RUNS: {NUM_RUNS}\n")
    f.write(f"CORRELATION_FACTOR: {CORRELATION_FACTOR}\n")

# ---------------------------------------------
# SECTION 2: AVS Reward & Slashing Assignments
# ---------------------------------------------
# AVSs are grouped into tiers by risk/reward. Higher reward = higher potential slashing loss.
reward_tiers = {
    "low": (0.0025, 0.004),    # +0.25–0.40% APR, low risk
    "medium": (0.004, 0.007),  # +0.40–0.70% APR, moderate risk
    "high": (0.007, 0.01)      # +0.70–1.00% APR, high risk
}

avs_reward_boosts = {}         # Dict: AVS ID → reward boost
avs_slash_penalties = {}       # Dict: AVS ID → penalty percentage (max 100%)

# Assign each AVS a reward and a proportional slashing penalty
for avs_id in range(NUM_AVS):
    tier = random.choice(list(reward_tiers.keys()))
    reward = round(np.random.uniform(*reward_tiers[tier]), 4)
    penalty = round(reward * np.random.uniform(6, 12), 4)  # Higher APR = higher risk
(proportional to reward)
    avs_reward_boosts[avs_id] = reward
    avs_slash_penalties[avs_id] = min(penalty, 1.0)        # Cap penalty at 100% of stake

# ---------------------------------------------
# SECTION 3: Operator Risk Profiles
# ---------------------------------------------
# Defines how many AVSs each operator engages with, based on risk appetite.
# More aggressive = more AVSs = more risk exposure.
RISK_PROFILES = {
    "conservative": {"prob": 0.4, "avs_pct_range": (0.01, 0.05)},
    "moderate": {"prob": 0.35, "avs_pct_range": (0.03, 0.10)},
```

```
    "aggressive": {"prob": 0.2, "avs_pct_range": (0.05, 0.25)},
    "ultra_aggressive": {"prob": 0.05, "avs_pct_range": (0.10, 1.00)}
}

# Generate operator population with assigned risk profiles and AVS selections
def generate_operators():
    operators = []
    risk_keys = list(RISK_PROFILES.keys())
    risk_probs = [RISK_PROFILES[k]["prob"] for k in risk_keys]
    for i in range(NUM_OPERATORS):
        profile = np.random.choice(risk_keys, p=risk_probs)
        avs_pct_range = RISK_PROFILES[profile]["avs_pct_range"]
        avs_percent = np.random.uniform(*avs_pct_range)
        num_avs = max(1, int(NUM_AVS * avs_percent))
        assigned_avs = random.sample(range(NUM_AVS), min(num_avs, NUM_AVS))
        operators.append({"id": i, "profile": profile, "avs": assigned_avs})
    return operators

# ---------------------------------------------
# SECTION 4: Core Simulation Logic
# ---------------------------------------------
# Simulates operator earnings or losses depending on scenario and exposure
def simulate_event(scenario, operators):
    # Set of AVSs that fail (based on scenario type)
    if scenario == "none":
        slashed_avss = set()
    elif scenario == "single":
        slashed_avss = {random.choice(range(NUM_AVS))}
    elif scenario == "cascade":
        slashed_avss = set(random.sample(range(NUM_AVS), 5))
    elif scenario == "panic":
        slashed_avss = set(random.sample(range(NUM_AVS), 20))

    earnings_data = []
    for op in operators:
        reward_bonus = sum([avs_reward_boosts[avs] for avs in op["avs"]])
        restaked_apr = ETH_APR + reward_bonus

        # Count how many of operator's AVSs are in the failed set
        failed_avs_count = sum(1 for avs in op["avs"] if avs in slashed_avss)

        # Apply slashing penalty if any of operator's AVSs are in the failed set
        if failed_avs_count > 0:
            # Base penalty calculation
            base_penalty = sum([avs_slash_penalties[avs] for avs in op["avs"] if avs in
slashed_avss])

            # Apply correlation factor for multiple failed AVSs
            if failed_avs_count > 1:
                # Increase penalty by correlation factor for each additional failed AVS
                correlation_multiplier = 1 + (CORRELATION_FACTOR * (failed_avs_count - 1))
                penalty = base_penalty * correlation_multiplier
            else:
                penalty = base_penalty

            penalty = min(penalty, 1.0)  # Cap penalty at 100%
            operator_earnings = -penalty  # Operator loses value
            slashed = True
        else:
            operator_earnings = restaked_apr
            slashed = False

        staker_apr = round(operator_earnings * 0.85, 4)  # Assumption: Delegators get 85% of
earnings

```

```
        earnings_data.append({
            "profile": op["profile"],
            "avs_count": len(op["avs"]),
            "failed_avs_count": failed_avs_count,  # Add this to track correlation effects
            "earnings": round(operator_earnings, 4),
            "delegator_apr": staker_apr,
            "slashed": slashed
        })

    return pd.DataFrame(earnings_data)

# ------------------------------------------
# SECTION 5: Monte Carlo Execution
# ------------------------------------------
aggregated_results = {scenario: [] for scenario in SCENARIOS}
summary_stats = []

for scenario in tqdm(SCENARIOS, desc="Running all scenarios"):
    for i in tqdm(range(NUM_RUNS), desc=f"Simulating {scenario}", leave=False):
        ops = generate_operators()
        df = simulate_event(scenario, ops)
        df["iteration"] = i + 1  # Track which run this was from
        aggregated_results[scenario].append(df)

# ------------------------------------------
# SECTION 6: Metrics and Summary Statistics
# ------------------------------------------
for scenario in tqdm(SCENARIOS, desc="Processing results"):
    combined = pd.concat(aggregated_results[scenario])
    # Extra safety check to ensure output directory exists
    if not os.path.exists(results_dir):
        os.makedirs(results_dir)
    combined.to_csv(f"{results_dir}/eigenlayer_{scenario}_full_output.csv", index=False)

    stats = {
        "Scenario": scenario,
        "Description": SCENARIO_LABELS[scenario],
        "% Slashed": round(100 * combined["slashed"].mean(), 2),
        "Avg Operator APR": round(combined["earnings"].mean(), 4),
        "Median Operator APR": round(combined["earnings"].median(), 4),
        "Avg Delegator APR": round(combined["delegator_apr"].mean(), 4),
        "Std Dev (Operator APR)": round(combined["earnings"].std(), 4),
        "Min APR": round(combined["earnings"].min(), 4),
        "Max APR": round(combined["earnings"].max(), 4),
        "Q1 APR": round(combined["earnings"].quantile(0.25), 4),
        "Q3 APR": round(combined["earnings"].quantile(0.75), 4)
    }

    # Add per-profile slashing rates to stats
    slashed_by_profile = combined[combined["slashed"]].groupby("profile").size().to_dict()
    total_by_profile = combined.groupby("profile").size().to_dict()
    for profile in total_by_profile:
        pct = 100 * slashed_by_profile.get(profile, 0) / total_by_profile[profile]
        stats[f"% Slashed ({profile})"] = round(pct, 2)

    summary_stats.append(stats)

summary_df = pd.DataFrame(summary_stats)
summary_df.to_csv(f"{results_dir}/eigenlayer_summary_statistics.csv", index=False)

# Print summary stats to console for quick review (sorted by % Slashed for clarity)
print("\n=== Summary Statistics ===")
print(summary_df.sort_values(by="% Slashed", ascending=False).to_string(index=False))

# ------------------------------------------
```

```
# SECTION 7: Visualizations
# ----------------------------------------
# Define shared color palette to ensure consistency across plots
profile_palette = {
    "conservative": "#1f77b4",  # Blue
    "moderate": "#2ca02c",      # Green
    "aggressive": "#ff7f0e",    # Orange
    "ultra_aggressive": "#d62728"  # Red
}

# Define consistent order for risk profiles
risk_profile_order = ["conservative", "moderate", "aggressive", "ultra_aggressive"]

# Create consistent legends
custom_legend = [
    Patch(color=profile_palette["conservative"], label="Conservative (1-5% of AVSs)"),
    Patch(color=profile_palette["moderate"], label="Moderate (3-10% of AVSs)"),
    Patch(color=profile_palette["aggressive"], label="Aggressive (5-25% of AVSs)"),
    Patch(color=profile_palette["ultra_aggressive"], label="Ultra-Aggressive (10-100% of AVSs)")
]

# Determine global min and max APR for consistent y-axis ranges
global_min_apr = min(df["earnings"].min() for df in aggregated_results.values() for df in df)
global_max_apr = max(df["earnings"].max() for df in aggregated_results.values() for df in df)

# Function for consistent boxplots
def create_consistent_boxplot(scenario, df, output_path):
    plt.figure(figsize=(10, 6))

    # Create a new DataFrame with reordered profiles to ensure consistent ordering
    plot_data = []
    for profile in risk_profile_order:
        profile_data = df[df["profile"] == profile]
        if not profile_data.empty:
            plot_data.append(profile_data)

    if plot_data:
        plot_df = pd.concat(plot_data)

        # Create boxplot with consistent order
        sns.boxplot(x="profile", y="earnings", hue="profile", data=plot_df,
                    palette=profile_palette, order=risk_profile_order, dodge=False, legend=False)

        plt.title(f"APR by Risk Profile – {scenario.capitalize()} Scenario")
        plt.xlabel("Operator Risk Profile")
        plt.ylabel("Operator APR")
        plt.ylim(global_min_apr, global_max_apr)  # Consistent y-axis
        plt.grid(True)

        # Place legend outside plot
        plt.legend(handles=custom_legend, title="Operator Risk Profile",
                   bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0.)

        plt.tight_layout()
        plt.savefig(output_path, bbox_inches="tight")
        plt.close()

# Function for faceted boxplot grid across scenarios
def create_faceted_apr_boxplots(output_path):
    scenario_order = ["none", "single", "cascade", "panic"]
    scenario_labels = {
        "none": "No AVS Failure (Base Case)",
        "single": "1 AVS Failure (Isolated Incident)",
        "cascade": "5 AVS Failure (Correlated Slashing)",
        "panic": "20 AVS Failure (Extreme Network Event)"
```

```
    }

    fig, axes = plt.subplots(2, 2, figsize=(16, 12))
    axes = axes.flatten()

    for idx, scenario in enumerate(scenario_order):
        ax = axes[idx]
        df = pd.concat(aggregated_results[scenario])
        df["profile"] = pd.Categorical(df["profile"], categories=risk_profile_order, ordered=True)

        sns.boxplot(
            x="profile", y="earnings", hue="profile", data=df, ax=ax,
            order=risk_profile_order, palette=profile_palette, dodge=False, legend=False
        )
        ax.set_title(scenario_labels[scenario])
        ax.set_xlabel("Risk Profile")
        ax.set_ylabel("Operator APR")
        ax.set_ylim(global_min_apr, global_max_apr)
        ax.grid(True)

    fig.suptitle("Operator APR Distribution by Risk Profile and Scenario", fontsize=16)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.savefig(output_path)
    plt.close()

# Generate the faceted APR boxplot across scenarios
create_faceted_apr_boxplots(f"{visuals_dir}/apr_boxplot_faceted_by_scenario.png")

# Function for consistent histograms
def create_consistent_histogram(scenario, df, output_path):
    plt.figure(figsize=(10, 6))

    # Create histogram with consistent bins and range
    sns.histplot(df["earnings"], bins=50)

    plt.title(f"Operator Earnings Distribution - {scenario.capitalize()} Scenario")
    plt.xlabel("Operator APR")
    plt.ylabel("Number of Operators")
    plt.xlim(global_min_apr, global_max_apr)
    plt.grid(True)
    plt.tight_layout()
    # Extra safety check to ensure visuals directory exists
    if not os.path.exists(visuals_dir):
        os.makedirs(visuals_dir)
    plt.savefig(output_path)
    plt.close()

# Function for facet grid of histograms
def create_facet_histograms(output_path):
    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # Consistent order of scenarios
    scenario_order = ["none", "single", "cascade", "panic"]
    scenario_labels = {
        "none": "No AVS Failure (Base Case)",
        "single": "1 AVS Failure (Isolated Incident)",
        "cascade": "5 AVS Failure (Correlated Slashing)",
        "panic": "20 AVS Failure (Extreme Network Event)"
    }

    for idx, scenario in enumerate(scenario_order):
        ax = axes[idx // 2, idx % 2]
        data = pd.concat(aggregated_results[scenario])

        sns.histplot(data["earnings"], bins=50, ax=ax)
```

```
376.
377.         ax.set_title(f"{scenario_labels[scenario]}")
378.         ax.set_xlabel("Operator APR")
379.         ax.set_ylabel("Count")
380.         ax.set_xlim(global_min_apr, global_max_apr)
381.         ax.grid(True)
382.
383.     plt.tight_layout()
384.     # Extra safety check to ensure visuals directory exists
385.     if not os.path.exists(visuals_dir):
386.         os.makedirs(visuals_dir)
387.     plt.savefig(output_path)
388.     plt.close()
389.
390. # Generate individual histograms
391. for scenario in SCENARIOS:
392.     df = pd.concat(aggregated_results[scenario])
393.     create_consistent_histogram(scenario, df, f"{visuals_dir}/earnings_histogram_{scenario}.png")
394.
395. # Generate facet grid of histograms
396. create_facet_histograms(f"{visuals_dir}/facet_histograms_all_scenarios.png")
397.
398. # Generate boxplots by risk profile
399. for scenario in SCENARIOS:
400.     df = pd.concat(aggregated_results[scenario])
401.     create_consistent_boxplot(scenario, df, f"{visuals_dir}/apr_by_risk_profile_{scenario}.png")
402.
403. # Function to create summary statistics table with improved formatting
404. def plot_summary_table(df, output_path):
405.     # Rename long column names to prevent overlap
406.     df = df.rename(columns={
407.         "Avg Operator APR": "Avg Op APR",
408.         "Median Operator APR": "Med Op APR",
409.         "Avg Delegator APR": "Avg Del APR",
410.         "Std Dev (Operator APR)": "Std Dev APR",
411.         "% Slashed (aggressive)": "% Slashed (aggr.)",
412.         "% Slashed (ultra_aggressive)": "% Slashed (ultra)",
413.         "% Slashed (moderate)": "% Slashed (mod.)",
414.         "% Slashed (conservative)": "% Slashed (cons.)"
415.     })
416.
417.     # Consistent ordering of rows based on scenario severity
418.     scenario_order = ["none", "single", "cascade", "panic"]
419.     df = df.set_index("Scenario").loc[scenario_order].reset_index()
420.
421.     # Create a more readable description column
422.     description_map = {
423.         "No AVS failure – base case": "No failures (base case)",
424.         "1 AVS failure – isolated incident": "Single AVS failure",
425.         "5 AVS failure – correlated slashing": "5 AVS failures (correlated)",
426.         "20 AVS failure – extreme network event": "20 AVS failures (extreme)"
427.     }
428.     df["Description"] = df["Description"].map(lambda x: description_map.get(x, x))
429.
430.     fig, ax = plt.subplots(figsize=(20, 6))
431.     ax.axis("tight")
432.     ax.axis("off")
433.
434.     # Create the table with improved formatting
435.     table = ax.table(
436.         cellText=df.round(3).values,
437.         colLabels=df.columns,
438.         cellLoc='center',
439.         loc='center'
440.     )
```

```
441.
442.     table.auto_set_font_size(False)
443.     table.set_fontsize(10)
444.     table.scale(1.5, 1.4)  # Slightly wider
445.
446.     # Style improvements for readability
447.     for (row, col), cell in table.get_celld().items():
448.         if row == 0:  # Header row
449.             cell.set_text_props(weight='bold', wrap=True)
450.             cell.set_height(0.15)  # Taller header cells for wrapped text
451.         if col == 0 or col == 1:  # Scenario and description columns
452.             cell.set_text_props(ha='left')
453.             cell.set_width(0.15)
454.
455.     plt.title("Exhibit: Summary Statistics Across Scenarios", fontsize=14, pad=10)
456.     plt.savefig(output_path, bbox_inches="tight", dpi=300)
457.     plt.close()
458.
459. # Create summary statistics table
460. plot_summary_table(summary_df, f"{visuals_dir}/summary_statistics_table.png")
461.
462. # ---------------------------------------------
463. # SECTION 7.1: Correlation Factor Impact Analysis - with Improved Data Collection
464. # ---------------------------------------------
465.
466. def analyze_correlation_impact():
467.     """Analyze and visualize the impact of the correlation factor on slashing severity"""
468.
469.     print("\n=== Correlation Factor Impact Analysis ===")
470.
471.     # Process ALL scenarios to collect any available data
472.     correlation_data = []
473.
474.     # Consistent scenario order and colors
475.     scenario_order = ["single", "cascade", "panic"]
476.     scenario_colors = {
477.         "single": "#ff9f1c",  # Light orange
478.         "cascade": "#2ec4b6",  # Teal
479.         "panic": "#e71d36"     # Red
480.     }
481.     scenario_labels = {
482.         "single": "1 AVS failure – isolated incident",
483.         "cascade": "5 AVS failure – correlated slashing",
484.         "panic": "20 AVS failure – extreme network event"
485.     }
486.
487.     # First collect all available data points from all scenarios
488.     for scenario in scenario_order:
489.         combined = pd.concat(aggregated_results[scenario])
490.
491.         # Look at slashed operators
492.         slashed_ops = combined[combined["slashed"]]
493.
494.         # Group by number of failed AVSs
495.         if not slashed_ops.empty:
496.             by_failed_count = slashed_ops.groupby("failed_avs_count").agg({
497.                 "earnings": ["mean", "min", "count"],
498.                 "profile": "count"
499.             })
500.
501.             # Format for plotting
502.             for count, row in by_failed_count.iterrows():
503.                 correlation_data.append({
504.                     "scenario": scenario,
505.                     "failed_avs_count": count,
```

```
                    "avg_loss": -row[("earnings", "mean")],  # Convert to positive loss value
                    "max_loss": -row[("earnings", "min")],
                    "operator_count": row[("profile", "count")]
                })

    if correlation_data:
        corr_df = pd.DataFrame(correlation_data)

        # Create the enhanced visualization with explanatory annotation
        plt.figure(figsize=(12, 7))

        # Calculate expected penalty based on correlation factor for theoretical curve
        # This will show how losses should scale with the correlation factor
        max_failed_avs = 20
        theoretical_x = np.arange(1, max_failed_avs + 1)

        # Base penalty for a single AVS failure
        # Find this from the data if available, otherwise use a reasonable value
        base_penalty = 0.05  # Starting with assumption of 5% loss for 1 AVS

        # Find the actual base penalty from each scenario if available
        for scenario in scenario_order:
            scenario_data = corr_df[corr_df["scenario"] == scenario]
            if not scenario_data.empty and 1 in scenario_data["failed_avs_count"].values:
                base_penalty_val = scenario_data[scenario_data["failed_avs_count"] ==
1]["avg_loss"].values[0]
                if scenario == "single":  # Use single scenario as benchmark if available
                    base_penalty = base_penalty_val

        # Calculate theoretical curve based on correlation factor
        theoretical_y = []
        for n in theoretical_x:
            if n == 1:
                penalty = base_penalty
            else:
                # Apply correlation factor multiplicatively
                penalty = min(base_penalty * (1 + CORRELATION_FACTOR * (n - 1)), 1.0)
            theoretical_y.append(penalty)

        # Plot theoretical curve
        plt.plot(theoretical_x, theoretical_y, 'k--', alpha=0.5,
                 label=f"Theoretical (correlation factor: {CORRELATION_FACTOR})")

        # Plot actual data for each scenario
        for scenario in scenario_order:
            scenario_data = corr_df[corr_df["scenario"] == scenario]

            # Only plot if there's data (and more than one data point ideally)
            if not scenario_data.empty and len(scenario_data) > 0:
                # Sort by failed_avs_count to ensure proper line connections
                scenario_data = scenario_data.sort_values("failed_avs_count")

                plt.plot(scenario_data["failed_avs_count"],
                         scenario_data["avg_loss"],
                         marker='o',
                         linewidth=2,
                         color=scenario_colors[scenario],
                         label=scenario_labels[scenario])

                # Add explanation for missing data if needed
                if scenario == "single" and len(scenario_data) <= 1:
                    plt.annotate(
                        "Single failure scenario typically\nonly affects 1 AVS per operator",
                        xy=(2, 0.1),
                        xytext=(3, 0.2),
```

```
                        arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2")
                    )

                if scenario == "cascade" and max(scenario_data["failed_avs_count"]) < 5:
                    plt.annotate(
                        "Cascade scenario rarely affects\nmore than a few AVSs per operator",
                        xy=(4, 0.3),
                        xytext=(6, 0.4),
                        arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2")
                    )

        plt.xlabel("Number of Failed AVSs for a Single Operator")
        plt.ylabel("Average Loss (negative APR)")
        plt.title("Impact of Multiple AVS Failures on Slashing Severity")
        plt.grid(True)

        handles, labels = plt.gca().get_legend_handles_labels()
        if labels:
            plt.legend(handles=handles, labels=labels)


        # Add explanation of correlation factor
        plt.figtext(0.5, 0.01,
                    f"Correlation Factor ({CORRELATION_FACTOR}): When multiple AVSs fail simultaneously, each additional AVS\n" +
                    f"increases the slashing penalty by {CORRELATION_FACTOR*100}% multiplicatively.",
                    ha="center", fontsize=10, bbox={"facecolor":"white", "alpha":0.5, "pad":5})

        plt.tight_layout(rect=[0, 0.05, 1, 0.95])  # Make room for the explanation text
        plt.savefig(f"{visuals_dir}/correlation_factor_impact.png")
        plt.close()

        # Save the data for reference
        corr_df.to_csv(f"{results_dir}/correlation_impact_analysis.csv", index=False)

        # Print summary to terminal for traceability
        print(f"Correlation impact chart saved to: {visuals_dir}/correlation_factor_impact.png")
        print(f"Raw data saved to: {results_dir}/correlation_impact_analysis.csv")
        print(f"Correlation data points collected: {len(corr_df)}")
        print(f"Max observed failed AVSs in one scenario: {corr_df['failed_avs_count'].max()}")

        return corr_df
    else:
        print("No correlation data available for analysis.")
        return None

# Call the correlation analysis function
correlation_df = analyze_correlation_impact()

# Operator APR Boxplot by Scenario and Risk Profile
# Combine all scenario DataFrames (no sampling)
full_frames = []
for scenario, data in aggregated_results.items():
    df = pd.concat(data)
    df["scenario"] = scenario.capitalize()
    full_frames.append(df)

combined_full_df = pd.concat(full_frames, ignore_index=True)
combined_full_df["earnings_percent"] = combined_full_df["earnings"] * 100

# Define risk profile order for consistency
profile_order = ["conservative", "moderate", "aggressive", "ultra_aggressive"]

# Generate and save boxplot
```

```
633. plt.figure(figsize=(16, 10))
634. ax = sns.boxplot(
635.     data=combined_full_df,
636.     x="profile",
637.     y="earnings_percent",
638.     hue="scenario",
639.     order=profile_order
640. )
641.
642. handles, labels = ax.get_legend_handles_labels()
643. plt.legend(handles=handles, labels=labels, title="Failure Scenario", loc="upper right")
644.
645. plt.title("Operator APR Distribution by Risk Profile and Scenario")
646. plt.xlabel("Validator Risk Profile")
647. plt.ylabel("Annual Percentage Rate (APR %)")
648. plt.grid(True)
649. plt.tight_layout()
650.
651. # Save figure to visualization directory
652. output_path = os.path.join(visuals_dir, "operator_apr_boxplot_by_scenario_full.png")
653. # Extra safety check to ensure visuals directory exists
654. if not os.path.exists(visuals_dir):
655.     os.makedirs(visuals_dir)
656. plt.savefig(output_path)
657. plt.close()
658.
659. # ---------------------------------------------
660. # SECTION 7.2 Risk Exposure Analysis - with Improved Risk Tolerance Explanation
661. # ---------------------------------------------
662. def analyze_risk_thresholds():
663.     """Analyze results to determine recommended maximum AVS exposure levels"""
664.
665.     print("\n=== Risk Exposure Analysis ===")
666.
667.     # Create a DataFrame to analyze slashing risk by AVS count
668.     risk_by_avs_count = {}
669.
670.     for scenario in SCENARIOS:
671.         combined = pd.concat(aggregated_results[scenario])
672.
673.         # Calculate average slashing rate by AVS count
674.         avs_group = combined.groupby("avs_count")
675.         avs_counts = avs_group["slashed"].mean() * 100
676.
677.         # Also calculate average APR by AVS count
678.         avg_apr = avs_group["earnings"].mean()
679.
680.         # Combine into a DataFrame
681.         scenario_df = pd.DataFrame({
682.             "slashing_pct": avs_counts,
683.             "avg_apr": avg_apr
684.         })
685.
686.         # Store results
687.         risk_by_avs_count[scenario] = scenario_df
688.
689.     # Combine all scenarios into one DataFrame for easier analysis
690.     all_data = []
691.
692.     # Consistent scenario order
693.     scenario_order = ["none", "single", "cascade", "panic"]
694.     scenario_colors = {
695.         "none": "#4d8fac",     # Blue
696.         "single": "#ff9f1c",   # Light orange
697.         "cascade": "#2ec4b6",  # Teal
```

```
        "panic": "#e71d36"     # Red
    }
    scenario_labels = {
        "none": "No AVS failure – base case",
        "single": "1 AVS failure – isolated incident",
        "cascade": "5 AVS failure – correlated slashing",
        "panic": "20 AVS failure – extreme network event"
    }

    for scenario in scenario_order:
        df = risk_by_avs_count[scenario]
        for avs_count, row in df.iterrows():
            all_data.append({
                "scenario": scenario,
                "avs_count": avs_count,
                "slashing_pct": row["slashing_pct"],
                "avg_apr": row["avg_apr"]
            })

    risk_analysis_df = pd.DataFrame(all_data)
    risk_analysis_df.to_csv(f"{results_dir}/risk_by_avs_count.csv", index=False)

    # Find thresholds where risk becomes unacceptable
    # Define risk tolerance levels with descriptions
    risk_tolerance = {
        "low": {"value": 5.0, "description": "Conservative validators (5% max slashing
probability)"},
        "medium": {"value": 15.0, "description": "Moderate risk tolerance (15% max slashing
probability)"},
        "high": {"value": 30.0, "description": "Aggressive risk appetite (30% max slashing
probability)"}
    }

    thresholds = {}

    # For each scenario (except "none"), find max AVS count for each risk tolerance
    for scenario in ['single', 'cascade', 'panic']:
        thresholds[scenario] = {}
        scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]

        for tolerance, info in risk_tolerance.items():
            max_pct = info["value"]
            # Find maximum AVS count where risk is below tolerance threshold
            below_threshold = scenario_data[scenario_data["slashing_pct"] <= max_pct]
            if not below_threshold.empty:
                max_safe_count = below_threshold["avs_count"].max()
            else:
                max_safe_count = 0
            thresholds[scenario][tolerance] = max_safe_count

    # Print recommendations
    print("\nRecommended Maximum AVS Exposure Guidelines:")
    for tolerance, info in risk_tolerance.items():
        print(f"\n{info['description']}:")
        for scenario in ['single', 'cascade', 'panic']:
            print(f"  - {scenario_labels[scenario]}: ≤ {thresholds[scenario][tolerance]} AVSs")

    # Create visualization of risk vs. AVS count for different scenarios
    plt.figure(figsize=(12, 7))

    # Plot each scenario line
    for scenario in scenario_order:
        scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]
        plt.plot(scenario_data["avs_count"],
                 scenario_data["slashing_pct"],
```

```
                marker='o',
                linewidth=2,
                color=scenario_colors[scenario],
                label=scenario_labels[scenario])

    # Add horizontal lines for risk tolerance levels with improved visuals
    tolerance_colors = {
        "low": "#a2d5f2",      # Light blue
        "medium": "#07689f",   # Medium blue
        "high": "#0a1931"      # Dark blue
    }

    # Add risk tolerance horizontal lines with clearer labels
    for tolerance, info in risk_tolerance.items():
        plt.axhline(y=info["value"],
                    linestyle='--',
                    alpha=0.7,
                    color=tolerance_colors[tolerance],
                    linewidth=2)
        # Add annotations directly on the lines for better clarity
        plt.text(95, info["value"] + 1,
                info["description"],
                color=tolerance_colors[tolerance],
                fontweight='bold',
                bbox=dict(facecolor='white', alpha=0.7, boxstyle='round,pad=0.5'))

    plt.xlabel('Number of AVSs Validated')
    plt.ylabel('Probability of Getting Slashed (%)')
    plt.title('Slashing Risk by Number of AVSs Validated')
    plt.grid(True)
    plt.legend(loc='upper left')

    # Add explanation text
    plt.figtext(0.5, 0.01,
                "Horizontal lines represent different risk tolerance levels. The intersection
points with scenario lines\n" +
                "indicate the maximum number of AVSs that can be validated within each risk
tolerance.",
                ha="center", fontsize=10, bbox={"facecolor":"white", "alpha":0.5, "pad":5})

    plt.tight_layout(rect=[0, 0.05, 1, 0.95])  # Make room for the explanation text
    plt.savefig(f"{visuals_dir}/risk_exposure_guidelines.png")
    plt.close()

    # Risk-Return Analysis: Plot APR vs Risk - IMPROVED
    plt.figure(figsize=(12, 7))

    # Plot each scenario with improved visibility
    for scenario in scenario_order:
        scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]
        plt.scatter(scenario_data["slashing_pct"],
                    scenario_data["avg_apr"],
                    s=80,
                    alpha=0.7,
                    color=scenario_colors[scenario],
                    label=scenario_labels[scenario])

    # Annotate strategic AVS counts for clarity
    key_avs_counts = [5, 10, 25, 50, 75, 100]

    for scenario in scenario_order:
        scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]
        for _, row in scenario_data.iterrows():
            if row["avs_count"] in key_avs_counts:
                plt.annotate(f"{int(row['avs_count'])} AVSs",
```

```
                    (row["slashing_pct"], row["avg_apr"]),
                     xytext=(5, 5),
                     textcoords="offset points",
                    fontsize=8,
                    bbox=dict(boxstyle="round,pad=0.3", fc="white", alpha=0.7)
                )

        plt.axhline(y=0, linestyle='-', color='black', alpha=0.3)
        plt.xlabel('Risk (Probability of Getting Slashed %)')
        plt.ylabel('Return (Average Operator APR)')
        plt.title('Risk-Return Profile by AVS Count and Scenario')
        plt.grid(True)

    # Handle legend more safely to avoid warnings
    handles, labels = plt.gca().get_legend_handles_labels()
    if labels:
        plt.legend(handles=handles, labels=labels)

    # Add explanation text
    plt.figtext(0.5, 0.01,
            "This chart shows the risk-return tradeoff for different AVS counts. Points above the horizontal line (y=0)\n"
            "represent positive expected returns, while points below indicate expected losses.",
            ha="center", fontsize=10, bbox={"facecolor": "white", "alpha": 0.5, "pad": 5})

    plt.tight_layout(rect=[0, 0.05, 1, 0.95])  # Make room for the explanation text
    plt.savefig(f"{visuals_dir}/risk_return_profile.png")
    plt.close()

    return thresholds, risk_analysis_df

# Call the risk analysis function
thresholds, risk_analysis_df = analyze_risk_thresholds()

# ------------------------------------------------
# END OF SIMULATION CODE
# ------------------------------------------------
# Show location of saved results so user can quickly find them
print(f"\n All simulation results saved to: {results_dir}")

# Close any remaining figures
plt.close('all')

# Confirm completion in terminal
print("Simulation Complete")

# ------------------------------------------------
# END OF FILE
```