

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

By Benjamin Erickson

Abstract

When Ethereum, the second-largest blockchain, transitioned from Proof-of-Work (PoW) to Proof-of-Stake (PoS) in 2022, it enhanced the network's energy efficiency, security, and scalability. This was a significant change, from a consensus mechanism that relied on energy-intensive mining to one that now secures the network through a layer of trusted validators who stake cryptocurrency. Blockchains are decentralized, meaning there is no central authority, so consensus, or trust and agreement, are the foundation of the network's security.

Ethereum's change to the PoS model, introduced a key weakness – capital inefficiency. Ethereum's currency, ETH, remains locked once staked, and unavailable for other economic purposes. EigenLayer, a middleware protocol that connects Ethereum to decentralized applications (DApps), addresses this problem by introducing a restaking model. This novel approach allows Ethereum validators to reuse staked ETH to secure other external services. In exchange for participating, validators have the opportunity to receive increased staking rewards, an incentive that builds trust and draws more validators to the network. Restaking not only increases capital efficiency for validators and developers, but adds a shared layer of security that DApps can tap into.

However, restaking does not come without system-wide risks and validator penalties. This paper examines those consequences, questioning whether EigenLayer's restaking model represents a sustainable improvement in capital efficiency or a potential threat to Ethereum's long-term security.

Using a custom-built simulation of 1,000 validators across four correlated failure scenarios, the paper evaluates validator behavior, yield distribution, and cascading slashing risk. The results show that while restaking improves capital efficiency, it also introduces compounding risks, including validator overexposure, staking concentration, and governance centralization – patterns that resemble collateral reuse and leverage in traditional finance. The paper concludes by evaluating EigenLayer's efforts towards implementing safeguards and presenting original proposals to improve its resilience, including adaptive risk scoring, a decentralized slashing insurance fund, tiered slashing mechanisms, and a transparent governance dashboard. If EigenLayer continues to implement such safeguards and evolve its risk management framework, its restaking model has the potential to significantly expand crypto-economic capital efficiency for Ethereum validators and developers without fundamentally undermining its trust or security model.

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 are able to 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).³

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

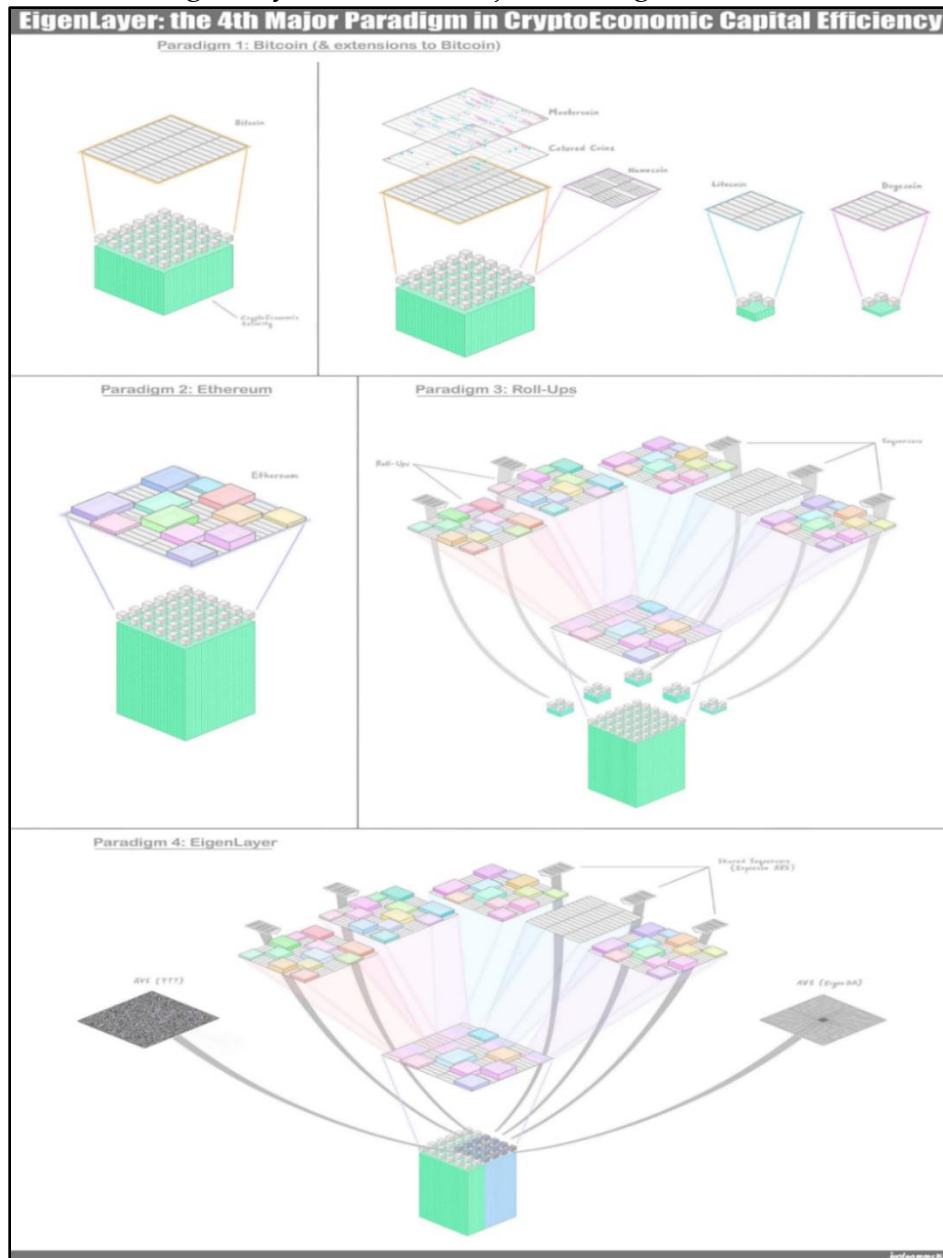
² EigenLayer. (2024). *EigenLayer whitepaper: Re-staking consensus and expanding cryptoeconomic security*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

³ McKinney, J. (2023). *EigenLayer explained: The fourth paradigm in crypto-economic capital efficiency*. <https://medium.com/@jordanmckinney/eigenlayer-explained-7f82adcb40ce>

Previous paradigms in crypto-economic capital efficiency have evolved from:

1. **First Paradigm (Bitcoin):** Pure Proof-of-Work with no staking
2. **Second Paradigm (Ethereum PoS):** Direct staking with locked capital
3. **Third Paradigm (Liquid Staking):** Liquid staking tokens that represent staked assets
4. **Fourth Paradigm (Restaking):** Reusing staked assets to secure multiple protocols simultaneously

Exhibit 1: EigenLayer – The 4th Major Paradigm⁴



⁴ McKinney, J. (2023). *EigenLayer explained: The fourth paradigm in crypto-economic capital efficiency*. <https://medium.com/@jordanmckinney/eigenlayer-explained-7f82adcb40ce>

Unlike traditional PoS and liquid staking models, 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.⁵

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.

⁵ Durvasula, K., & Roughgarden, T. (2024). *Cascading slashing in shared-security proof-of-stake systems*. arXiv. <https://arxiv.org/abs/2411.11789>

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 "Actively Validated Services" (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: Chainlink currently operates as a decentralized oracle network with its own validators called "node operators." If Chainlink were to integrate with EigenLayer, it could leverage Ethereum's larger validator pool instead of maintaining a separate system.⁶

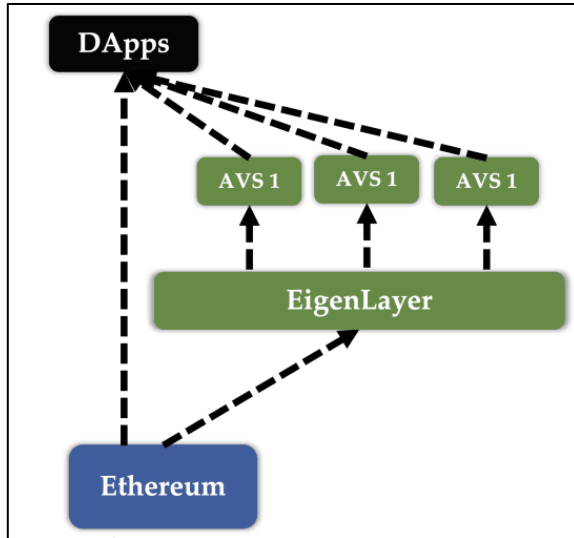
Benefits might include:

- 1. Improved Security:** Instead of relying solely on its own node operators, Chainlink could tap into Ethereum's extensive validator set, potentially enhancing its security through a larger, more diverse group of validators.
- 2. Reduced Costs:** Less overhead for maintaining a custom validator set.
- 3. Higher Capital Efficiency:** Ethereum validators could earn Chainlink rewards without staking LINK separately.

⁶ Chainlink. (2023). *Chainlink staking v0.1 overview: Securing oracle networks with staked LINK*. <https://chain.link/staking>

4. **Lower Barriers to Entry:** New services similar to Chainlink could launch without building validator networks from scratch.

Exhibit 2: EigenLayer Framework⁷



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

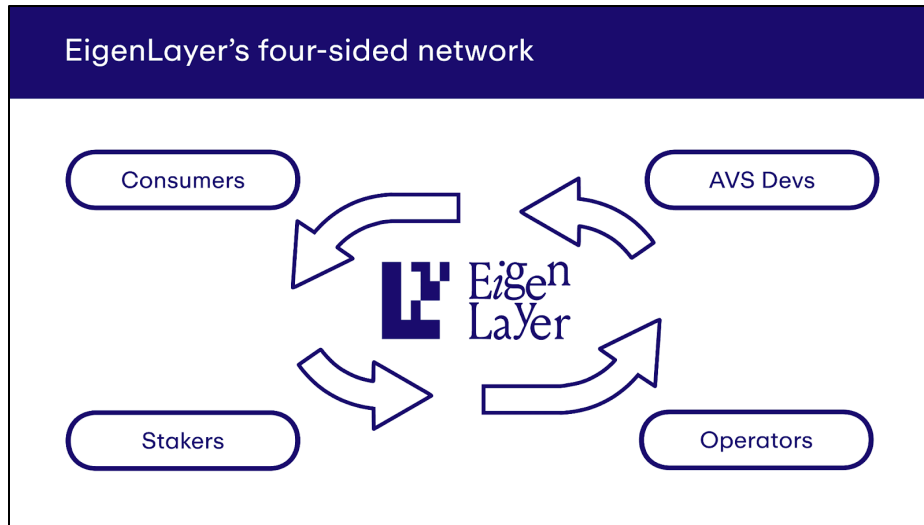
EigenLayer introduces a unique process called restaking, allowing Ethereum validators to reuse previously staked ETH as collateral to secure multiple blockchain services simultaneously. Under Ethereum's current Proof-of-Stake (PoS) model, validators lock up 32 ETH (worth over \$50,000 as of early 2025) 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.

⁷ EigenLayer. (2024). *EigenLayer whitepaper: Re-staking consensus and expanding cryptoeconomic security*. 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 3: EigenLayer's Four-Sided Network⁸

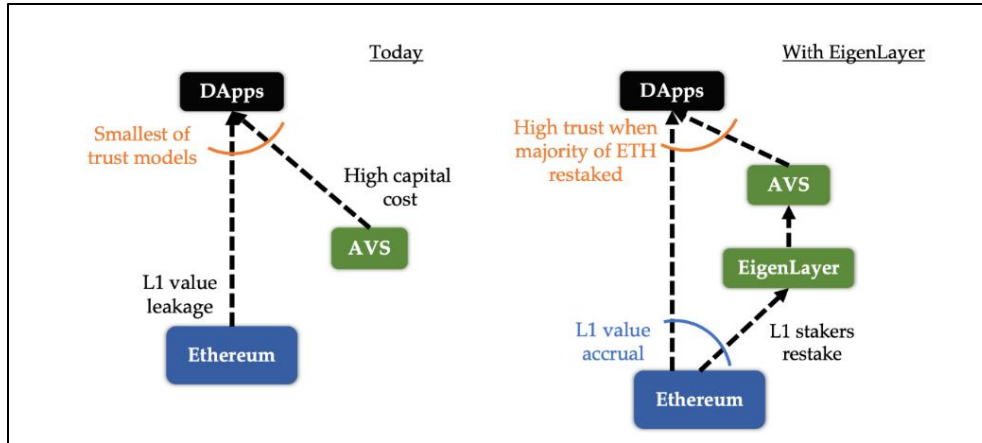


Restaking allows validators to amplify earnings by receiving rewards not only from Ethereum's base protocol, but also from multiple external services simultaneously. For example, a validator staking through EigenLayer has the opportunity to earn Ethereum's base staking APY between 3-5%, plus an additional 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 bootstrap early adoption while AVS rewards remain minimal, EigenLayer has launched a native governance token, EIGEN. As of early 2025, approximately 4% of the token supply is being distributed through a programmatic incentives model. Restakers and operators receive allocations proportional to the amount and duration of stake delegated across supported AVSs. While EIGEN is non-transferable at launch and not used for slashing, it is expected to play a significant role in protocol governance and future enforcement of subjective fault claims.⁹

⁸EigenLayer. *EigenLayer Documentation*. Accessed April 2025. <https://docs.eigenlayer.xyz/>

⁹EigenLayer. (2024). *EigenLayer whitepaper: Re-staking consensus and expanding cryptoeconomic security*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

Exhibit 4: EigenLayer's Shared Security Model¹⁰Shared 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 structure. 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 lose a portion of their staked ETH. By expanding slashing penalties to include behaviors relevant to AVSs, EigenLayer significantly increases the security of multiple protocols.

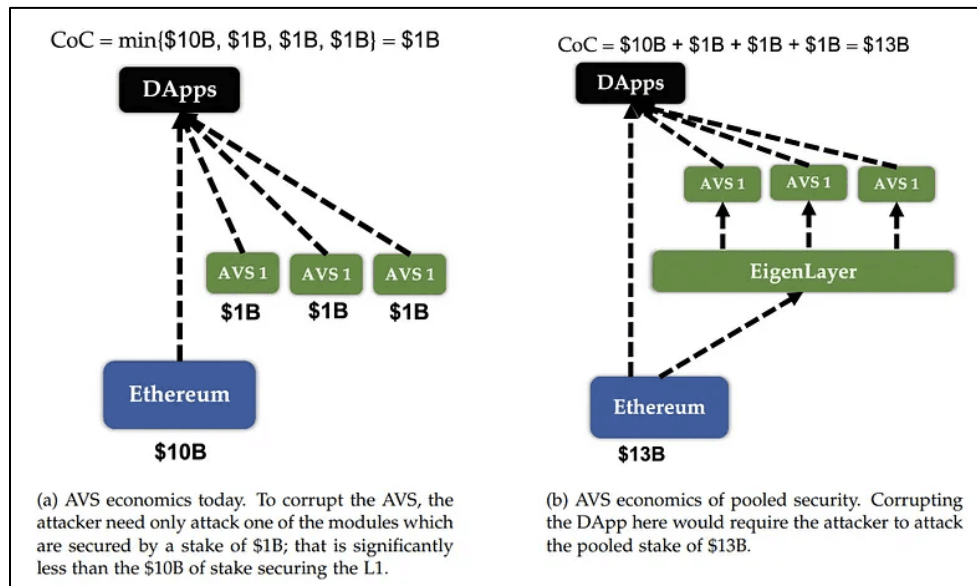
The shared security model solves a key challenge in blockchain development—how protocols establish trust without building their own validator networks. New blockchain protocols typically struggle to attract enough validators to ensure their security. EigenLayer solves this by allowing new protocols to 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 economically irrational in most scenarios.

¹⁰ EigenLayer. (2024). *EigenLayer whitepaper: Re-staking consensus and expanding cryptoeconomic security*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611dec6e562ad.pdf

The diagram shows how EigenLayer functions as a bridge between Ethereum validators and DApps, creating pooled security that improves capital efficiency, increases the potential to earn greater rewards, and provides a cost-effective and efficient security layer for AVSs.

Exhibit 5: Pooled Security of EigenLayer¹¹



Restaking Risks and Governance Challenges

Additional risks and penalties come with restaking, particularly in slashing and governance, which can potentially undermine Ethereum's security:

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 turn into a 15% loss. If slashed below Ethereum's 32 ETH threshold, validators are forced to exit, losing rewards and weakening network security.¹²

¹¹ EigenLayer. (2024). *EigenLayer whitepaper: Re-staking consensus and expanding cryptoeconomic security*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf

¹² ConsenSys. (2024). *Understanding slashing in Ethereum staking: Its importance and consequences*. <https://consensys.io/blog/understanding-slashing-in-ethereum-staking-its-importance-and-consequences>

When many validators share exposure, one failure can trigger widespread slashing. This type of collateral overextension resembles traditional financial risk, as elaborated in Section III.

2. Dual Slashing

This situation occurs when validators are penalized by both Ethereum and one or more AVSs for the same or related misbehavior. Validators operating within EigenLayer are subject to two distinct rule sets: Ethereum's core consensus rules and the unique 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: if a dominant AVS fails, many validators may be affected at once.

4. Governance Risk: Complexity and Centralization

EigenLayer's restaking model introduces a complex governance landscape where multiple AVSs depend on the same validator set, each with unique slashing conditions and operational requirements. Coordinating these diverse interests demands robust governance mechanisms, but without safeguards, this coordination can lead to centralization. Economically dominant AVSs may exert outsized influence, shaping policies in their favor and marginalizing smaller participants. This dynamic threatens to erode Ethereum's decentralization by concentrating control over security policy in the hands of a few powerful stakeholders, undermining the open, permissionless principles that define DeFi.

EigenLayer's restaking and shared security framework boosts capital efficiency but introduces key risks. Validators face cascading and dual slashing, where a single failure can lead to multiple penalties and forced exits. Staking concentration around high-reward AVSs and governance centralization can further weaken the system. The inherent risk of overextended collateralization in restaking echoes the weaknesses seen in traditional finance.¹³

¹³ EigenLayer. (2024). *Introducing EigenGov: Part 1*. <https://docs.eigenlayer.xyz/eigengov/governance/introducing-eigengov-part-1>

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.¹⁴

¹⁴ CoinMarketCap. (2024). *The ultimate guide to Ethereum liquid staking*. <https://coinmarketcap.com/academy/article/the-ultimate-guide-to-ethereum-liquid-staking>

Exhibit 6: Platform Comparisons - EigenLayer vs. Lido, Rocket Pool, and Centralized Services ¹⁵

| 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 protocol-native rewards from AVSs. This opportunity, however, comes with added 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.

¹⁵Sources: 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 7: Where to Deploy stETH—Comparing Returns, Risks, and Ecosystem Impact ¹⁶

| 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 pre-2008—financially productive, but systemically fragile.

¹⁶Sources: Based on data from Kiln.fi, EigenLayer docs, Cointelegraph, Lido, Rocket Pool, DeFiLlama, and public LRT documentation as of April 2025.

Traditional Finance Parallel: Rehypothection and Systemic Fragility

EigenLayer's restaking model mirrors the financial practice of rehypothection, 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.¹⁷

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.¹⁸

¹⁷ Investopedia. (2024). *Rehypothection*. <https://www.investopedia.com/terms/r/rehypothection.asp>

¹⁸ 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 appetite 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 condition

For each scenario, comprehensive metrics were tracked including operator APR distribution, slashing probability, and system-wide economic impact.¹⁹

¹⁹ 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 8: Slashing Probability by Validator Risk Profile (Cascade Scenario)²⁰

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

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

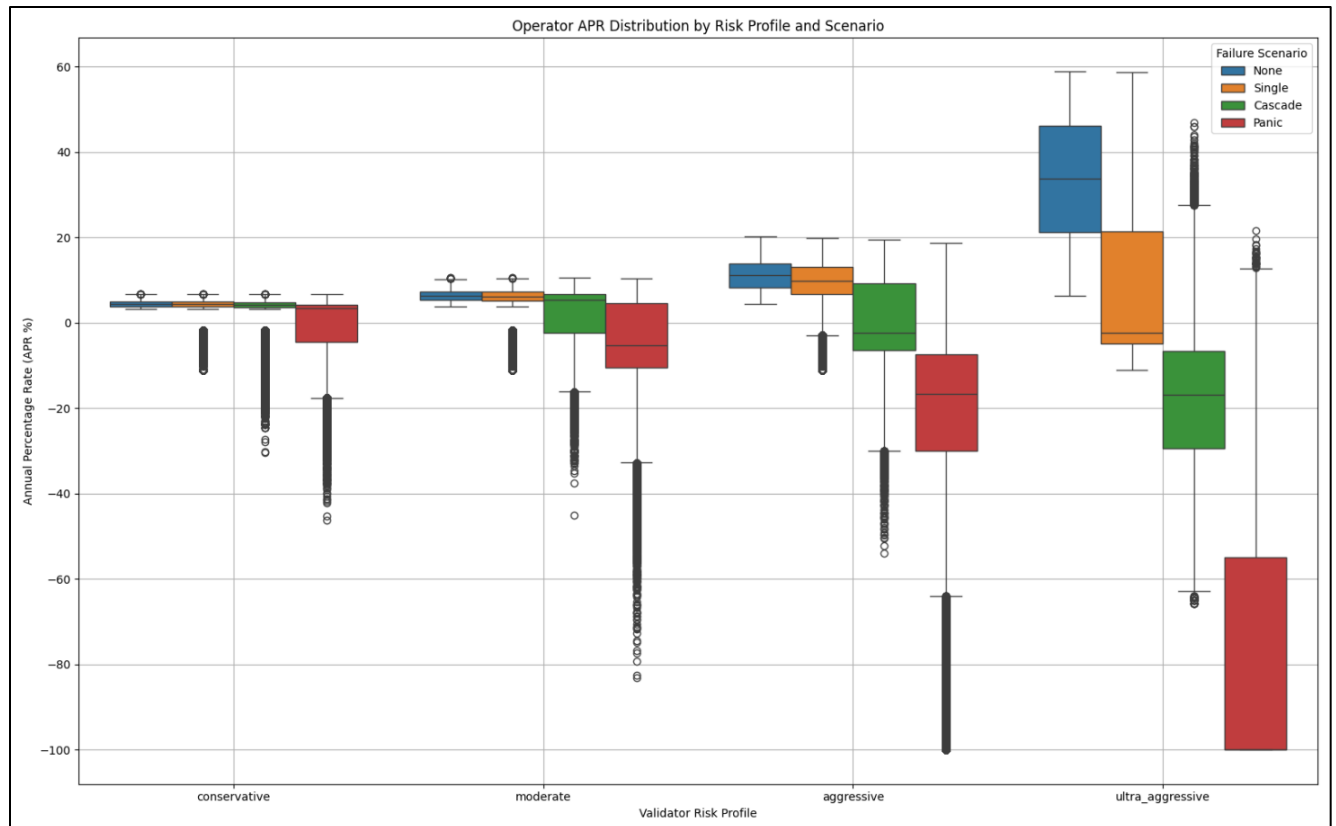
These results demonstrate the need for validators to carefully manage their AVS exposure. With the implementation of features like "Unique Stake Allocation" that EigenLayer is developing, validators could isolate their risk across different AVSs, preventing the cascade effect and maintaining the capital 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 in place, the impact of even a severe market dislocation could be contained, preserving Ethereum's security while still allowing for the capital efficiency benefits of restaking under normal conditions.

Exhibit 9: APR Outcomes by Scenario and Validator Type



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: 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

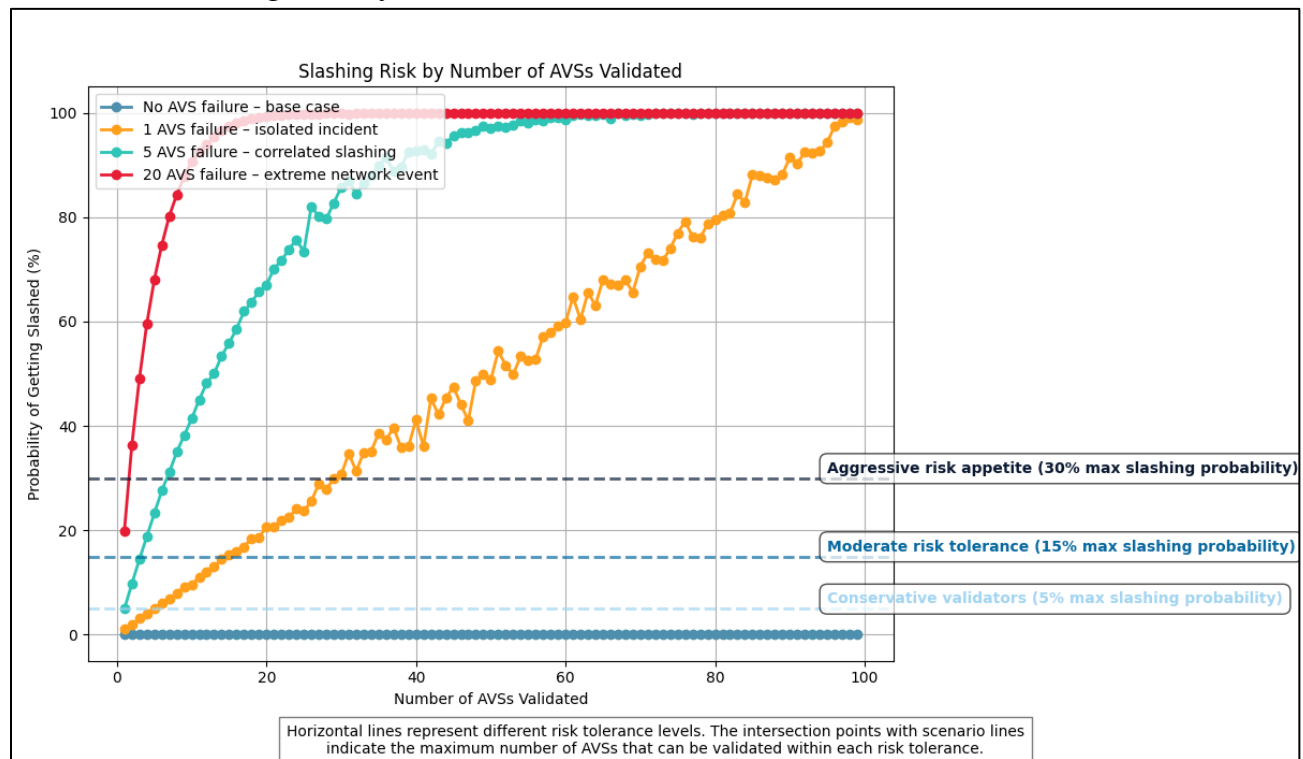
To provide practical guidance for validators, the relationship between AVS participation count and slashing probability was analyzed across different risk tolerance levels. This analysis reveals clear thresholds for safe participation in EigenLayer's ecosystem. The probability of experiencing at least 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 10: Maximum Safe Participation Thresholds²¹

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

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

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

Exhibit 11: Slashing Risk by Number of AVSs Validated²²

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 12: Validator Slashing and Projected ETH Network Stake Reduction²³

| 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 the majority of 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 efficacy 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 EigenLayer's risk dynamics, 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, information asymmetry 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

²⁵Erickson, 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 enhancements. 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 associated with slashing and validator overexposure, EigenLayer employs several strategies:

- **Unique Stake Allocation and Operator Sets**

Validators can assign specific portions of their stake to individual Actively Validated Services (AVSs), ensuring that slashing in one AVS does not compromise their entire stake.

- **Pooled Security Model**

By aggregating over \$7 billion in restaked ETH, EigenLayer increases the economic cost of attacks, making malicious actions financially unviable.²⁶

- **Compensatory Mechanisms**

AVSs can acquire slashing redistribution rights, allowing for the return of funds to affected parties in the event of honest validator errors, thereby transforming slashing from a purely punitive measure to a partially recoverable one. For instance, a bridge could acquire redistribution rights for \$25 million; if a validator it relies on is slashed, the penalty could be routed back to cover those losses.

- **Anti-Slashing Infrastructure**

Tools such as Puffer's anti-slasher libraries reduce the technical barriers for validators and help prevent accidental slashable behavior, promoting broader participation and decentralization.²⁷

²⁶ EigenLayer. (2024). *Risk management overview*. <https://docs.eigenlayer.xyz/eigenlayer/security/slashing-and-risk-management>

²⁷ ConsensSys. (2024). *Understanding slashing in Ethereum staking: Its importance and consequences*. <https://consensys.io/blog/understanding-slashing-in-ethereum-staking-its-importance-and-consequences>

Countering Centralization Risks

To prevent centralization and promote equitable participation:

- **Decentralized Governance Council (EigenGov)**
EigenLayer has established EigenGov, a network of councils composed of domain experts empowered by EIGEN token holders to make protocol decisions, ensuring that governance remains decentralized and representative.
- **Support for Lightweight AVSs**
By promoting AVSs with minimal hardware requirements, such as EigenDA, EigenLayer lowers entry barriers for validators, encouraging participation from a diverse set of operators.
- **Task Delegation**
Validators with limited resources can delegate tasks to more capable operators without forfeiting their rewards, maintaining inclusivity within the network.
- **Incentives for Solo Validators**
EigenLayer offers reward multipliers and reduced minimum stake thresholds for solo validators, recognizing their role in enhancing network resilience and reducing correlated failure risks. These incentives help ensure solo validators not only enter the ecosystem, but continue to contribute to its decentralization and resilience over time.²⁸

Managing Validator Integrity and Conflicts of Interest

To prevent validator misconduct and align economic incentives:

- **Technical Enforcement Mechanisms**
The use of Trusted Execution Environments (TEEs) and standardized anti-slasher libraries ensures that validator actions comply with AVS-specific requirements, limiting the potential for misconduct.
- **Economic Incentives and Constraints**
Validators are required to post collateral, aligning their financial interests with the protocol's health. Staking caps and liquidity buffers further limit exposure and mitigate the impact of malicious behavior.

²⁸ Galaxy Digital. (2024). *Restaking risks and rewards*. <https://www.galaxy.com/research/>

- **Transparency and Community Oversight**

Real-time dashboards, third-party audits, and open governance discussions facilitate transparency, allowing stakeholders to monitor validator performance and respond to emerging risks effectively.

The Role of EIGEN in Subjective Enforcement

The EIGEN token serves a pivotal role in governance and subjective enforcement within EigenLayer:

- **Governance Participation**

EIGEN holders can vote on critical protocol policies, including AVS approvals and dispute resolution procedures, ensuring that governance decisions reflect the community's collective will.

- **Handling Intersubjective Faults**

EIGEN enables the community to address faults that cannot be resolved through cryptographic means alone, such as oracle discrepancies, by facilitating coordinated responses and, if necessary, initiating token forks to penalize malicious actors. Together, ETH and EIGEN form a dual-token security model: ETH provides the economic collateral for objective enforcement, while EIGEN facilitates subjective dispute resolution and governance.²⁹

Proposals for Strengthening Safeguards

While EigenLayer's current safeguards offer a strong foundation, the following enhancements are proposed to further reduce systemic risk and improve resilience:

1. **Adaptive Risk Scoring for AVSs**

AVSs should be assigned dynamic risk scores based on performance metrics such as uptime, validator slashing history, technical complexity, and dispute rates. These scores would be made public through a transparent dashboard, allowing validators to filter and rank AVSs by risk-adjusted APR. This would reduce information asymmetry and help validators tailor their exposure to their individual risk tolerance levels.

²⁹ EigenLayer. (2024). *Introducing EigenGov: Part 1*. <https://docs.eigenlayer.xyz/eigengov/governance/introducing-eigengov-part-1>

2. Decentralized Slashing Insurance Fund

Establish a protocol-native insurance fund governed by EIGEN holders. Validators and AVSs would pay risk-based premiums into the fund, which would compensate honest validators in the event of large-scale slashing caused by infrastructure failures or unforeseen AVS collapse. Unlike third-party insurance models, this mutualized system could be transparently managed and adjusted by on-chain governance.

3. Tiered Slashing Penalties Based on Behavior

Introduce a graduated penalty structure for validators that distinguishes between minor mistakes and malicious activity. First-time infractions could result in warnings or small slashes, while repeated or severe violations would invoke harsher penalties. This nuanced approach promotes validator retention, encourages honest participation, and avoids over-penalizing accidental behavior.

4. On-Chain Governance Transparency Dashboard

Develop a publicly accessible governance explorer that displays live data on all governance activities—AVS onboarding, slashing appeals, insurance fund proposals, and EIGEN council votes. This dashboard would serve as both a transparency mechanism and an educational tool, helping build trust in EigenLayer's governance and enabling informed community participation.

While EigenLayer introduces systemic risks, its current safeguards and the proposed improvements offer a realistic path to managing them. When these technical, economic, and governance protections work in tandem, EigenLayer strengthens Ethereum's modular security without compromising decentralization, achieving the thesis of this paper: improved capital efficiency without fundamentally undermining security.

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

- Beaconcha.in. (2025). *Ethereum validator data explorer*. <https://beaconcha.in>
- Chainlink. (2023). *Chainlink staking v0.1 overview: Securing oracle networks with staked LINK*. <https://chain.link/staking>
- Coinbase. (2024). *Guide to EigenLayer: Restaking explained*. <https://www.coinbase.com/learn/crypto-basics/what-is-eigenlayer>
- CoinMarketCap. (2024). *The ultimate guide to Ethereum liquid staking*. <https://coinmarketcap.com/academy/article/the-ultimate-guide-to-ethereum-liquid-staking>
- ConsensSys. (2024). *What is EigenLayer and why it matters*. <https://consensys.io/blog/eigenlayer/>
- ConsensSys. (2024). *Understanding slashing in Ethereum staking: Its importance and consequences*. <https://consensys.io/blog/understanding-slashing-in-ethereum-staking-its-importance-and-consequences>
- DAIC Capital. (2024). *EigenLayer restaking protocol overview*. <https://daic.capital/blog/eigen-layer-restaking-protocol>
- Darshan, G. (2024). *Drop #62: EigenLayer's restaking revolution*. <https://darshang.substack.com/p/drop-62-eigen-layer>
- Durvasula, K., & Roughgarden, T. (2024). *Cascading slashing in shared-security proof-of-stake systems*. arXiv. <https://arxiv.org/abs/2411.11789>
- Erickson, B. (2025). *EigenLayer simulation repository*. GitHub. https://github.com/bserickson/eigenlayer_simulation
- EigenLayer. (2024). *EigenLayer whitepaper: Re-staking consensus and expanding cryptoeconomic security*. https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf
- EigenLayer. (2024). *Introducing EigenGov: Part 1*. <https://docs.eigenlayer.xyz/eigengov/governance/introducing-eigengov-part-1>
- EigenLayer. (2024). *Risk management overview*. <https://docs.eigenlayer.xyz/eigenlayer/security/slashing-and-risk-management>
- EigenLayer. *EigenLayer Documentation*. Accessed April 2025. <https://docs.eigenlayer.xyz/>
- Ethereum Foundation. (2024). *Staking on Ethereum*. <https://ethereum.org/en/staking/>
- Financial Times. (2020). *Rehypothecation: Wall Street's dangerous secret*. <https://www.ft.com>
- Galaxy Digital. (2024). *Restaking risks and rewards*. <https://www.galaxy.com/research/>
- Investopedia. (2024). *Rehypothecation*. <https://www.investopedia.com/terms/r/rehypothecation.asp>
- McKinney, J. (2023). *EigenLayer explained: The fourth paradigm in crypto-economic capital efficiency*. Medium. <https://medium.com/@jordanmckinney/eigenlayer-explained-7f82adcb40ce>
- OpenAI. (2024). *ChatGPT (March 14 version) [Large language model]*. <https://chat.openai.com>

Rocket Pool. (2024). *rETH mechanics and decentralized staking infrastructure*.

<https://docs.rocketpool.net>

Wikipedia contributors. (n.d.). *Financial crisis of 2007–2008*. Wikipedia.

https://en.wikipedia.org/wiki/Financial_crisis_of_2007%E2%80%932008

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

```

1. import numpy as np
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. from matplotlib.patches import Patch
5. import seaborn as sns
6. import random
7. from collections import defaultdict
8. import os
9. from tqdm import tqdm
10. from datetime import datetime
11. import warnings
12. import logging
13. logging.getLogger('matplotlib').setLevel(logging.ERROR)
14.
15. # -----
16. # EigenLayer Simulation
17. # -----
18. # Simulates economic outcomes for EigenLayer operators who restake ETH across AVSs (Actively
Validated Services).
19. # Monte Carlo simulation models economic risks (slashing) and returns (APR) under multiple failure
scenarios.
20. # Outputs include scenario-based CSVs, summary statistics, and high-quality visualizations for
analysis/reporting.
21.
22. # -----
23. # SECTION 0: Seed Control for Reproducibility
24. # -----
25. # Allows switching between reproducible results (SEED = 42) or randomized results using current
timestamp
26. USE_RANDOM_SEED = False          # Set to True to use a new seed every time you run, False for
fixed seed
27. SEED = int(datetime.now().timestamp()) if USE_RANDOM_SEED else 42
28. np.random.seed(SEED)             # Makes NumPy random outputs reproducible
29. random.seed(SEED)                # Makes Python random outputs reproducible
30.
31. # -----
32. # SECTION 1: Global Parameters and Initialization
33. # -----
34. NUM_OPERATORS = 1000             # Number of simulated operators staking via EigenLayer
35. NUM_AVS = 100                    # Number of Actively Validated Services (AVSs)
36. ETH_APR = 0.03                   # Base Ethereum staking return (APR)
37. NUM_RUNS = 1000                  # Monte Carlo iterations per scenario for robustness
38. CORRELATION_FACTOR = 0.15        # Increase slashing probability for each additional failed AVS
39.
40. # -----
41. # Print Simulation Overview to Terminal
42. # -----
43. # This provides a readable summary of the simulation parameters before execution,
44. # useful for understanding the run context, debugging, or replicating results later.
45. # Includes operator count, AVS count, APR assumption, iteration count, and seed.
46. print("=" * 60)
47. print("Running EigenLayer Operator Simulation")
48. print("- Simulates economic outcomes from restaking ETH across AVSs")
49. print("- Models operator rewards and slashing risks under 4 scenarios")
50. print(f"- Monte Carlo simulation with {NUM_OPERATORS} operators x {NUM_RUNS} iterations")
51. print(f"- {NUM_AVS} Actively Validated Services (AVSs) simulated")
52. print(f"- Assumes Ethereum base staking APR = {ETH_APR:.2%}")
53. print(f"- Using correlation factor = {CORRELATION_FACTOR} for recursive slashing")

```

```

54. print(f"- Simulation SEED: {SEED}")
55. print("=" * 60)
56. print("=" * 60)
57.
58. # -----
59. # Continue SECTION 1: Global Parameters and Initialization
60. # -----
61. # Four failure scenarios: increasing in severity
62. SCENARIOS = ["none", "single", "cascade", "panic"]
63. SCENARIO_LABELS = {
64.     "none": "No AVS failure - base case",
65.     "single": "1 AVS failure - isolated incident",
66.     "cascade": "5 AVS failure - correlated slashing",
67.     "panic": "20 AVS failure - extreme network event"
68. }
69.
70. # Create timestamped directory to store output
71. timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
72. results_dir = f"results_{timestamp}"
73. os.makedirs(results_dir, exist_ok=True)
74.
75. # Create subfolder for all generated visualizations
76. visuals_dir = os.path.join(results_dir, "visualizations")
77. os.makedirs(visuals_dir, exist_ok=True)
78.
79. # Save configuration parameters for reference and reproducibility
80. with open(f"{results_dir}/config.txt", "w") as f:
81.     f.write("=== Simulation Parameters ===\n")
82.     f.write(f"SEED: {SEED}\n")
83.     f.write(f"NUM_OPERATORS: {NUM_OPERATORS}\n")
84.     f.write(f"NUM_AVS: {NUM_AVS}\n")
85.     f.write(f"ETH_APR: {ETH_APR}\n")
86.     f.write(f"NUM_RUNS: {NUM_RUNS}\n")
87.     f.write(f"CORRELATION_FACTOR: {CORRELATION_FACTOR}\n")
88.
89. # -----
90. # SECTION 2: AVS Reward & Slashing Assignments
91. # -----
92. # AVSs are grouped into tiers by risk/reward. Higher reward = higher potential slashing loss.
93. reward_tiers = {
94.     "low": (0.0025, 0.004),    # +0.25-0.40% APR, low risk
95.     "medium": (0.004, 0.007), # +0.40-0.70% APR, moderate risk
96.     "high": (0.007, 0.01)     # +0.70-1.00% APR, high risk
97. }
98.
99. avs_reward_boosts = {}        # Dict: AVS ID → reward boost
100. avs_slash_penalties = {}     # Dict: AVS ID → penalty percentage (max 100%)
101.
102. # Assign each AVS a reward and a proportional slashing penalty
103. for avs_id in range(NUM_AVS):
104.     tier = random.choice(list(reward_tiers.keys()))
105.     reward = round(np.random.uniform(*reward_tiers[tier]), 4)
106.     penalty = round(reward * np.random.uniform(6, 12), 4) # Higher APR = higher risk
107.     avs_reward_boosts[avs_id] = reward
108.     avs_slash_penalties[avs_id] = min(penalty, 1.0)      # Cap penalty at 100% of stake
109.
110. # -----
111. # SECTION 3: Operator Risk Profiles
112. # -----
113. # Defines how many AVSs each operator engages with, based on risk appetite.
114. # More aggressive = more AVSs = more risk exposure.
115. RISK_PROFILES = {
116.     "conservative": {"prob": 0.4, "avs_pct_range": (0.01, 0.05)},
117.     "moderate": {"prob": 0.35, "avs_pct_range": (0.03, 0.10)},

```



```

118.     "aggressive": {"prob": 0.2, "avs_pct_range": (0.05, 0.25)},
119.     "ultra_aggressive": {"prob": 0.05, "avs_pct_range": (0.10, 1.00)}
120. }
121.
122. # Generate operator population with assigned risk profiles and AVS selections
123. def generate_operators():
124.     operators = []
125.     risk_keys = list(RISK_PROFILES.keys())
126.     risk_probs = [RISK_PROFILES[k]["prob"] for k in risk_keys]
127.     for i in range(NUM_OPERATORS):
128.         profile = np.random.choice(risk_keys, p=risk_probs)
129.         avs_pct_range = RISK_PROFILES[profile]["avs_pct_range"]
130.         avs_percent = np.random.uniform(*avs_pct_range)
131.         num_avs = max(1, int(NUM_AVS * avs_percent))
132.         assigned_avs = random.sample(range(NUM_AVS), min(num_avs, NUM_AVS))
133.         operators.append({"id": i, "profile": profile, "avs": assigned_avs})
134.     return operators
135.
136. # -----
137. # SECTION 4: Core Simulation Logic
138. # -----
139. # Simulates operator earnings or losses depending on scenario and exposure
140. def simulate_event(scenario, operators):
141.     # Set of AVSs that fail (based on scenario type)
142.     if scenario == "none":
143.         slashed_avss = set()
144.     elif scenario == "single":
145.         slashed_avss = {random.choice(range(NUM_AVS))}
146.     elif scenario == "cascade":
147.         slashed_avss = set(random.sample(range(NUM_AVS), 5))
148.     elif scenario == "panic":
149.         slashed_avss = set(random.sample(range(NUM_AVS), 20))
150.
151.     earnings_data = []
152.     for op in operators:
153.         reward_bonus = sum([avs_reward_boosts[avs] for avs in op["avs"]])
154.         restaked_apr = ETH_APR + reward_bonus
155.
156.         # Count how many of operator's AVSs are in the failed set
157.         failed_avs_count = sum(1 for avs in op["avs"] if avs in slashed_avss)
158.
159.         # Apply slashing penalty if any of operator's AVSs are in the failed set
160.         if failed_avs_count > 0:
161.             # Base penalty calculation
162.             base_penalty = sum([avs_slash_penalties[avs] for avs in op["avs"] if avs in
slashed_avss])
163.
164.             # Apply correlation factor for multiple failed AVSs
165.             if failed_avs_count > 1:
166.                 # Increase penalty by correlation factor for each additional failed AVS
167.                 correlation_multiplier = 1 + (CORRELATION_FACTOR * (failed_avs_count - 1))
168.                 penalty = base_penalty * correlation_multiplier
169.             else:
170.                 penalty = base_penalty
171.
172.             penalty = min(penalty, 1.0) # Cap penalty at 100%
173.             operator_earnings = -penalty # Operator loses value
174.             slashed = True
175.         else:
176.             operator_earnings = restaked_apr
177.             slashed = False
178.
179.     staker_apr = round(operator_earnings * 0.85, 4) # Assumption: Delegators get 85% of
earnings
180.

```

```

181.         earnings_data.append({
182.             "profile": op["profile"],
183.             "avs_count": len(op["avs"]),
184.             "failed_avs_count": failed_avs_count, # Add this to track correlation effects
185.             "earnings": round(operator_earnings, 4),
186.             "delegator_apr": staker_apr,
187.             "slashed": slashed
188.         })
189.
190.     return pd.DataFrame(earnings_data)
191.
192. # -----
193. # SECTION 5: Monte Carlo Execution
194. # -----
195. aggregated_results = {scenario: [] for scenario in SCENARIOS}
196. summary_stats = []
197.
198. for scenario in tqdm(SCENARIOS, desc="Running all scenarios"):
199.     for i in tqdm(range(NUM_RUNS), desc=f"Simulating {scenario}", leave=False):
200.         ops = generate_operators()
201.         df = simulate_event(scenario, ops)
202.         df["iteration"] = i + 1 # Track which run this was from
203.         aggregated_results[scenario].append(df)
204.
205. # -----
206. # SECTION 6: Metrics and Summary Statistics
207. # -----
208. for scenario in tqdm(SCENARIOS, desc="Processing results"):
209.     combined = pd.concat(aggregated_results[scenario])
210.     # Extra safety check to ensure output directory exists
211.     if not os.path.exists(results_dir):
212.         os.makedirs(results_dir)
213.     combined.to_csv(f"{results_dir}/eigenlayer_{scenario}_full_output.csv", index=False)
214.
215.     stats = {
216.         "Scenario": scenario,
217.         "Description": SCENARIO_LABELS[scenario],
218.         "% Slashed": round(100 * combined["slashed"].mean(), 2),
219.         "Avg Operator APR": round(combined["earnings"].mean(), 4),
220.         "Median Operator APR": round(combined["earnings"].median(), 4),
221.         "Avg Delegator APR": round(combined["delegator_apr"].mean(), 4),
222.         "Std Dev (Operator APR)": round(combined["earnings"].std(), 4),
223.         "Min APR": round(combined["earnings"].min(), 4),
224.         "Max APR": round(combined["earnings"].max(), 4),
225.         "Q1 APR": round(combined["earnings"].quantile(0.25), 4),
226.         "Q3 APR": round(combined["earnings"].quantile(0.75), 4)
227.     }
228.
229.     # Add per-profile slashing rates to stats
230.     slashed_by_profile = combined[combined["slashed"]].groupby("profile").size().to_dict()
231.     total_by_profile = combined.groupby("profile").size().to_dict()
232.     for profile in total_by_profile:
233.         pct = 100 * slashed_by_profile.get(profile, 0) / total_by_profile[profile]
234.         stats[f"% Slashed ({profile})"] = round(pct, 2)
235.
236.     summary_stats.append(stats)
237.
238. summary_df = pd.DataFrame(summary_stats)
239. summary_df.to_csv(f"{results_dir}/eigenlayer_summary_statistics.csv", index=False)
240.
241. # Print summary stats to console for quick review (sorted by % Slashed for clarity)
242. print("\n=== Summary Statistics ===")
243. print(summary_df.sort_values(by="% Slashed", ascending=False).to_string(index=False))
244.
245. # -----

```

```

246. # SECTION 7: Visualizations
247. # -----
248. # Define shared color palette to ensure consistency across plots
249. profile_palette = {
250.     "conservative": "#1f77b4", # Blue
251.     "moderate": "#2ca02c", # Green
252.     "aggressive": "#ff7f0e", # Orange
253.     "ultra_aggressive": "#d62728" # Red
254. }
255.
256. # Define consistent order for risk profiles
257. risk_profile_order = ["conservative", "moderate", "aggressive", "ultra_aggressive"]
258.
259. # Create consistent legends
260. custom_legend = [
261.     Patch(color=profile_palette["conservative"], label="Conservative (1-5% of AVSs)",
262.     Patch(color=profile_palette["moderate"], label="Moderate (3-10% of AVSs)",
263.     Patch(color=profile_palette["aggressive"], label="Aggressive (5-25% of AVSs)",
264.     Patch(color=profile_palette["ultra_aggressive"], label="Ultra-Aggressive (10-100% of AVSs)")
265. ]
266.
267. # Determine global min and max APR for consistent y-axis ranges
268. global_min_apr = min(df["earnings"].min() for df in aggregated_results.values() for df in df)
269. global_max_apr = max(df["earnings"].max() for df in aggregated_results.values() for df in df)
270.
271. # Function for consistent boxplots
272. def create_consistent_boxplot(scenario, df, output_path):
273.     plt.figure(figsize=(10, 6))
274.
275.     # Create a new DataFrame with reordered profiles to ensure consistent ordering
276.     plot_data = []
277.     for profile in risk_profile_order:
278.         profile_data = df[df["profile"] == profile]
279.         if not profile_data.empty:
280.             plot_data.append(profile_data)
281.
282.     if plot_data:
283.         plot_df = pd.concat(plot_data)
284.
285.         # Create boxplot with consistent order
286.         sns.boxplot(x="profile", y="earnings", hue="profile", data=plot_df,
287.             palette=profile_palette, order=risk_profile_order, dodge=False, legend=False)
288.
289.         plt.title(f"APR by Risk Profile - {scenario.capitalize()} Scenario")
290.         plt.xlabel("Operator Risk Profile")
291.         plt.ylabel("Operator APR")
292.         plt.ylim(global_min_apr, global_max_apr) # Consistent y-axis
293.         plt.grid(True)
294.
295.         # Place legend outside plot
296.         plt.legend(handles=custom_legend, title="Operator Risk Profile",
297.             bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0.)
298.
299.         plt.tight_layout()
300.         plt.savefig(output_path, bbox_inches="tight")
301.         plt.close()
302.
303. # Function for faceted boxplot grid across scenarios
304. def create_faceted_apr_boxplots(output_path):
305.     scenario_order = ["none", "single", "cascade", "panic"]
306.     scenario_labels = {
307.         "none": "No AVS Failure (Base Case)",
308.         "single": "1 AVS Failure (Isolated Incident)",
309.         "cascade": "5 AVS Failure (Correlated Slashing)",
310.         "panic": "20 AVS Failure (Extreme Network Event)"

```

```

311.     }
312.
313.     fig, axes = plt.subplots(2, 2, figsize=(16, 12))
314.     axes = axes.flatten()
315.
316.     for idx, scenario in enumerate(scenario_order):
317.         ax = axes[idx]
318.         df = pd.concat(aggregated_results[scenario])
319.         df["profile"] = pd.Categorical(df["profile"], categories=risk_profile_order, ordered=True)
320.
321.         sns.boxplot(
322.             x="profile", y="earnings", hue="profile", data=df, ax=ax,
323.             order=risk_profile_order, palette=profile_palette, dodge=False, legend=False
324.         )
325.         ax.set_title(scenario_labels[scenario])
326.         ax.set_xlabel("Risk Profile")
327.         ax.set_ylabel("Operator APR")
328.         ax.set_ylim(global_min_apr, global_max_apr)
329.         ax.grid(True)
330.
331.     fig.suptitle("Operator APR Distribution by Risk Profile and Scenario", fontsize=16)
332.     plt.tight_layout(rect=[0, 0.03, 1, 0.95])
333.     plt.savefig(output_path)
334.     plt.close()
335.
336. # Generate the faceted APR boxplot across scenarios
337. create_faceted_apr_boxplots(f"{visuals_dir}/apr_boxplot_faceted_by_scenario.png")
338.
339. # Function for consistent histograms
340. def create_consistent_histogram(scenario, df, output_path):
341.     plt.figure(figsize=(10, 6))
342.
343.     # Create histogram with consistent bins and range
344.     sns.histplot(df["earnings"], bins=50)
345.
346.     plt.title(f"Operator Earnings Distribution - {scenario.capitalize()} Scenario")
347.     plt.xlabel("Operator APR")
348.     plt.ylabel("Number of Operators")
349.     plt.xlim(global_min_apr, global_max_apr)
350.     plt.grid(True)
351.     plt.tight_layout()
352.     # Extra safety check to ensure visuals directory exists
353.     if not os.path.exists(visuals_dir):
354.         os.makedirs(visuals_dir)
355.     plt.savefig(output_path)
356.     plt.close()
357.
358. # Function for facet grid of histograms
359. def create_facet_histograms(output_path):
360.     fig, axes = plt.subplots(2, 2, figsize=(14, 10))
361.
362.     # Consistent order of scenarios
363.     scenario_order = ["none", "single", "cascade", "panic"]
364.     scenario_labels = {
365.         "none": "No AVS Failure (Base Case)",
366.         "single": "1 AVS Failure (Isolated Incident)",
367.         "cascade": "5 AVS Failure (Correlated Slashing)",
368.         "panic": "20 AVS Failure (Extreme Network Event)"
369.     }
370.
371.     for idx, scenario in enumerate(scenario_order):
372.         ax = axes[idx // 2, idx % 2]
373.         data = pd.concat(aggregated_results[scenario])
374.
375.         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,

```

```

506.         "avg_loss": -row[("earnings", "mean")], # Convert to positive loss value
507.         "max_loss": -row[("earnings", "min")],
508.         "operator_count": row[("profile", "count")]
509.     })
510.
511. if correlation_data:
512.     corr_df = pd.DataFrame(correlation_data)
513.
514.     # Create the enhanced visualization with explanatory annotation
515.     plt.figure(figsize=(12, 7))
516.
517.     # Calculate expected penalty based on correlation factor for theoretical curve
518.     # This will show how losses should scale with the correlation factor
519.     max_failed_avs = 20
520.     theoretical_x = np.arange(1, max_failed_avs + 1)
521.
522.     # Base penalty for a single AVS failure
523.     # Find this from the data if available, otherwise use a reasonable value
524.     base_penalty = 0.05 # Starting with assumption of 5% loss for 1 AVS
525.
526.     # Find the actual base penalty from each scenario if available
527.     for scenario in scenario_order:
528.         scenario_data = corr_df[corr_df["scenario"] == scenario]
529.         if not scenario_data.empty and 1 in scenario_data["failed_avs_count"].values:
530.             base_penalty_val = scenario_data[scenario_data["failed_avs_count"] ==
531. 1][["avg_loss"].values[0]
532.             if scenario == "single": # Use single scenario as benchmark if available
533.                 base_penalty = base_penalty_val
534.
535.     # Calculate theoretical curve based on correlation factor
536.     theoretical_y = []
537.     for n in theoretical_x:
538.         if n == 1:
539.             penalty = base_penalty
540.         else:
541.             # Apply correlation factor multiplicatively
542.             penalty = min(base_penalty * (1 + CORRELATION_FACTOR * (n - 1)), 1.0)
543.             theoretical_y.append(penalty)
544.
545.     # Plot theoretical curve
546.     plt.plot(theoretical_x, theoretical_y, 'k--', alpha=0.5,
547.              label=f"Theoretical (correlation factor: {CORRELATION_FACTOR})")
548.
549.     # Plot actual data for each scenario
550.     for scenario in scenario_order:
551.         scenario_data = corr_df[corr_df["scenario"] == scenario]
552.
553.         # Only plot if there's data (and more than one data point ideally)
554.         if not scenario_data.empty and len(scenario_data) > 0:
555.             # Sort by failed_avs_count to ensure proper line connections
556.             scenario_data = scenario_data.sort_values("failed_avs_count")
557.
558.             plt.plot(scenario_data["failed_avs_count"],
559.                      scenario_data["avg_loss"],
560.                      marker='o',
561.                      linewidth=2,
562.                      color=scenario_colors[scenario],
563.                      label=scenario_labels[scenario])
564.
565.     # Add explanation for missing data if needed
566.     if scenario == "single" and len(scenario_data) <= 1:
567.         plt.annotate(
568.             "Single failure scenario typically\nonly affects 1 AVS per operator",
569.             xy=(2, 0.1),
570.             xytext=(3, 0.2),

```



```

570.         arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2")
571.     )
572.
573.     if scenario == "cascade" and max(scenario_data["failed_avs_count"]) < 5:
574.         plt.annotate(
575.             "Cascade scenario rarely affects\nmore than a few AVSs per operator",
576.             xy=(4, 0.3),
577.             xytext=(6, 0.4),
578.             arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2")
579.         )
580.
581.     plt.xlabel("Number of Failed AVSs for a Single Operator")
582.     plt.ylabel("Average Loss (negative APR)")
583.     plt.title("Impact of Multiple AVS Failures on Slashing Severity")
584.     plt.grid(True)
585.
586.     handles, labels = plt.gca().get_legend_handles_labels()
587.     if labels:
588.         plt.legend(handles=handles, labels=labels)
589.
590.
591.     # Add explanation of correlation factor
592.     plt.figtext(0.5, 0.01,
593.         f"Correlation Factor ({CORRELATION_FACTOR}): When multiple AVSs fail
594. simultaneously, each additional AVS\n" +
595.         f"increases the slashing penalty by {CORRELATION_FACTOR*100}%
596. multiplicatively.",
597.         ha="center", fontsize=10, bbox={"facecolor":"white", "alpha":0.5, "pad":5})
598.
599.     plt.tight_layout(rect=[0, 0.05, 1, 0.95]) # Make room for the explanation text
600.     plt.savefig(f"{visuals_dir}/correlation_factor_impact.png")
601.     plt.close()
602.
603.     # Save the data for reference
604.     corr_df.to_csv(f"{results_dir}/correlation_impact_analysis.csv", index=False)
605.
606.     # Print summary to terminal for traceability
607.     print(f"Correlation impact chart saved to: {visuals_dir}/correlation_factor_impact.png")
608.     print(f"Raw data saved to: {results_dir}/correlation_impact_analysis.csv")
609.     print(f"Correlation data points collected: {len(corr_df)}")
610.     print(f"Max observed failed AVSs in one scenario: {corr_df['failed_avs_count'].max()}")
611.
612.     return corr_df
613. else:
614.     print("No correlation data available for analysis.")
615.     return None
616.
617. # Call the correlation analysis function
618. correlation_df = analyze_correlation_impact()
619.
620. # Operator APR Boxplot by Scenario and Risk Profile
621. # Combine all scenario DataFrames (no sampling)
622. full_frames = []
623. for scenario, data in aggregated_results.items():
624.     df = pd.concat(data)
625.     df["scenario"] = scenario.capitalize()
626.     full_frames.append(df)
627.
628. combined_full_df = pd.concat(full_frames, ignore_index=True)
629. combined_full_df["earnings_percent"] = combined_full_df["earnings"] * 100
630.
631. # Define risk profile order for consistency
632. profile_order = ["conservative", "moderate", "aggressive", "ultra_aggressive"]

```



```

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

```

```

698.         "panic": "#e71d36"      # Red
699.     }
700.     scenario_labels = {
701.         "none": "No AVS failure - base case",
702.         "single": "1 AVS failure - isolated incident",
703.         "cascade": "5 AVS failure - correlated slashing",
704.         "panic": "20 AVS failure - extreme network event"
705.     }
706.
707.     for scenario in scenario_order:
708.         df = risk_by_avs_count[scenario]
709.         for avs_count, row in df.iterrows():
710.             all_data.append({
711.                 "scenario": scenario,
712.                 "avs_count": avs_count,
713.                 "slashing_pct": row["slashing_pct"],
714.                 "avg_apr": row["avg_apr"]
715.             })
716.
717.     risk_analysis_df = pd.DataFrame(all_data)
718.     risk_analysis_df.to_csv(f"{results_dir}/risk_by_avs_count.csv", index=False)
719.
720.     # Find thresholds where risk becomes unacceptable
721.     # Define risk tolerance levels with descriptions
722.     risk_tolerance = {
723.         "low": {"value": 5.0, "description": "Conservative validators (5% max slashing
724.         probability)"},
725.         "medium": {"value": 15.0, "description": "Moderate risk tolerance (15% max slashing
726.         probability)"},
727.         "high": {"value": 30.0, "description": "Aggressive risk appetite (30% max slashing
728.         probability)"}
729.     }
730.
731.     thresholds = {}
732.
733.     # For each scenario (except "none"), find max AVS count for each risk tolerance
734.     for scenario in ['single', 'cascade', 'panic']:
735.         thresholds[scenario] = {}
736.         scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]
737.
738.         for tolerance, info in risk_tolerance.items():
739.             max_pct = info["value"]
740.             # Find maximum AVS count where risk is below tolerance threshold
741.             below_threshold = scenario_data[scenario_data["slashing_pct"] <= max_pct]
742.             if not below_threshold.empty:
743.                 max_safe_count = below_threshold["avs_count"].max()
744.             else:
745.                 max_safe_count = 0
746.             thresholds[scenario][tolerance] = max_safe_count
747.
748.     # Print recommendations
749.     print("\nRecommended Maximum AVS Exposure Guidelines:")
750.     for tolerance, info in risk_tolerance.items():
751.         print(f"\n{info['description']}")
752.         for scenario in ['single', 'cascade', 'panic']:
753.             print(f"    - {scenario_labels[scenario]}: ≤ {thresholds[scenario][tolerance]} AVSs")
754.
755.     # Create visualization of risk vs. AVS count for different scenarios
756.     plt.figure(figsize=(12, 7))
757.
758.     # Plot each scenario line
759.     for scenario in scenario_order:
760.         scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]
761.         plt.plot(scenario_data["avs_count"],
762.                 scenario_data["slashing_pct"],

```

```

760.         marker='o',
761.         linewidth=2,
762.         color=scenario_colors[scenario],
763.         label=scenario_labels[scenario])
764.
765.     # Add horizontal lines for risk tolerance levels with improved visuals
766.     tolerance_colors = {
767.         "low": "#a2d5f2",      # Light blue
768.         "medium": "#07689f",   # Medium blue
769.         "high": "#0a1931"     # Dark blue
770.     }
771.
772.     # Add risk tolerance horizontal lines with clearer labels
773.     for tolerance, info in risk_tolerance.items():
774.         plt.axhline(y=info["value"],
775.                     linestyle='--',
776.                     alpha=0.7,
777.                     color=tolerance_colors[tolerance],
778.                     linewidth=2)
779.         # Add annotations directly on the lines for better clarity
780.         plt.text(95, info["value"] + 1,
781.                  info["description"],
782.                  color=tolerance_colors[tolerance],
783.                  fontweight='bold',
784.                  bbox=dict(facecolor='white', alpha=0.7, boxstyle='round,pad=0.5'))
785.
786.     plt.xlabel('Number of AVSs Validated')
787.     plt.ylabel('Probability of Getting Slashed (%)')
788.     plt.title('Slashing Risk by Number of AVSs Validated')
789.     plt.grid(True)
790.     plt.legend(loc='upper left')
791.
792.     # Add explanation text
793.     plt.figtext(0.5, 0.01,
794.                 "Horizontal lines represent different risk tolerance levels. The intersection\n" +
795.                 "points with scenario lines\n" +
796.                 "indicate the maximum number of AVSs that can be validated within each risk\n" +
797.                 "tolerance.",
798.                 ha="center", fontsize=10, bbox={"facecolor": "white", "alpha": 0.5, "pad": 5})
799.
800.     plt.tight_layout(rect=[0, 0.05, 1, 0.95]) # Make room for the explanation text
801.     plt.savefig(f"{visuals_dir}/risk_exposure_guidelines.png")
802.     plt.close()
803.
804.     # Risk-Return Analysis: Plot APR vs Risk - IMPROVED
805.     plt.figure(figsize=(12, 7))
806.
807.     # Plot each scenario with improved visibility
808.     for scenario in scenario_order:
809.         scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]
810.         plt.scatter(scenario_data["slashing_pct"],
811.                     scenario_data["avg_apr"],
812.                     s=80,
813.                     alpha=0.7,
814.                     color=scenario_colors[scenario],
815.                     label=scenario_labels[scenario])
816.
817.     # Annotate strategic AVS counts for clarity
818.     key_avs_counts = [5, 10, 25, 50, 75, 100]
819.
820.     for scenario in scenario_order:
821.         scenario_data = risk_analysis_df[risk_analysis_df["scenario"] == scenario]
822.         for _, row in scenario_data.iterrows():
823.             if row["avs_count"] in key_avs_counts:
824.                 plt.annotate(f"{int(row['avs_count'])} AVSs",

```

```

823.             (row["slashing_pct"], row["avg_apr"]),
824.             xytext=(5, 5),
825.             textcoords="offset points",
826.             fontsize=8,
827.             bbox=dict(boxstyle="round,pad=0.3", fc="white", alpha=0.7)
828.         )
829.
830.     plt.axhline(y=0, linestyle='--', color='black', alpha=0.3)
831.     plt.xlabel('Risk (Probability of Getting Slashed %)')
832.     plt.ylabel('Return (Average Operator APR)')
833.     plt.title('Risk-Return Profile by AVS Count and Scenario')
834.     plt.grid(True)
835.
836.     # Handle legend more safely to avoid warnings
837.     handles, labels = plt.gca().get_legend_handles_labels()
838.     if labels:
839.         plt.legend(handles=handles, labels=labels)
840.
841.     # Add explanation text
842.     plt.figtext(0.5, 0.01,
843.                "This chart shows the risk-return tradeoff for different AVS counts. Points above the\n"
844.                "horizontal line (y=0)\n"
845.                "represent positive expected returns, while points below indicate expected losses.",
846.                ha="center", fontsize=10, bbox={"facecolor": "white", "alpha": 0.5, "pad": 5})
847.     plt.tight_layout(rect=[0, 0.05, 1, 0.95]) # Make room for the explanation text
848.     plt.savefig(f"{visuals_dir}/risk_return_profile.png")
849.     plt.close()
850.
851.     return thresholds, risk_analysis_df
852.
853. # Call the risk analysis function
854. thresholds, risk_analysis_df = analyze_risk_thresholds()
855.
856. # -----
857. # END OF SIMULATION CODE
858. # -----
859. # Show location of saved results so user can quickly find them
860. print(f"\n All simulation results saved to: {results_dir}")
861.
862. # Close any remaining figures
863. plt.close('all')
864.
865. # Confirm completion in terminal
866. print("Simulation Complete")
867.
868. # -----
869. # END OF FILE

```