An In-Depth Architectural Analysis of the Ethereum Name Service (ENS) Protocol

Introduction

The Ethereum Name Service (ENS) represents a foundational pillar of Web3 infrastructure, engineered to enhance the usability and accessibility of the decentralized web. At its core, ENS is a distributed, open, and extensible naming system built on the Ethereum blockchain.[1] Its primary function is to map human-readable names, such as

vitalik.eth, to machine-readable identifiers like Ethereum addresses, other cryptocurrency addresses, content hashes, and various forms of metadata.[1] This process transforms the often intimidating and error-prone experience of interacting with long hexadecimal strings—for example,

0x0b08dA7068b73A579Bd5E8a8290ff8afd37bc32A—into a simple, memorable, and more secure action.[4]

However, to categorize ENS as a mere "nickname generator" would be a significant understatement of its architectural depth and strategic importance. It is designed not to replace the internet's existing Domain Name System (DNS) but to complement and extend its capabilities into the decentralized realm.[1] By doing so, ENS provides a crucial service layer for digital identity, allowing users to consolidate their on-chain presence under a single, user-owned, and censorship-resistant name.[8] This name can function as a universal username, a payment address for multiple cryptocurrencies, and a pointer to decentralized websites, creating a cohesive and portable Web3 identity.[3]

This report provides a definitive technical resource for blockchain engineers, delivering an exhaustive analysis of the ENS protocol's architecture, core operational processes, and practical implementation. It will deconstruct the sophisticated system of smart contracts that govern the protocol, from the central Registry to the logic-defining Registrars and the data-serving Resolvers. Furthermore, it will explore the end-to-end name resolution lifecycle, advanced scalability solutions like off-chain and Layer 2 data retrieval, and the mechanisms

for integrating traditional DNS names. Through detailed explanations and practical code examples, this analysis aims to equip the reader with an expert-level understanding of the Ethereum Name Service and its pivotal role in the future of the decentralized internet.

Section 1: The Core Architectural Pillars of ENS

The resilience, flexibility, and extensibility of the Ethereum Name Service are direct results of its meticulously designed modular architecture. The protocol is not a monolithic entity but a system composed of three distinct and interoperable smart contract types: the Registry, Registrars, and Resolvers. This deliberate separation of concerns is a cornerstone of the protocol's design, creating a stable and secure core while enabling a permissionless environment for innovation at its periphery. Each component has a narrowly defined role, and their interaction forms the basis for all ENS operations.

1.1 The ENS Registry: The Single Source of Truth

At the absolute heart of the ENS protocol lies the ENS Registry, a single, central smart contract that functions as the definitive ledger for all registered names.[2] As specified in EIP-137, the Registry's design is intentionally minimalist to maximize security and long-term stability. Its primary and almost exclusive responsibility is to maintain a mapping from a unique identifier for each name—a 32-byte hash known as a

node—to a record containing three critical pieces of information:

1. **The Owner:** The Ethereum address that has control over the name's record in the Registry. The owner has the authority to transfer ownership, set the resolver, and create subdomains.[6]
2. **The Resolver:** The address of the smart contract responsible for translating the name into actual data (e.g., a cryptocurrency address or content hash). This level of indirection is what makes ENS so extensible.[6]
3. **The Time-To-Live (TTL):** A value that indicates how long clients should cache the name's resolution data, functioning as a hint to optimize performance.[6]

The Registry does not understand or interpret the human-readable names themselves; it deals exclusively with their hashed node representations. It also has no knowledge of the registration process or the rules for name allocation. Its function is purely administrative: to store and serve the authoritative pointers for who owns a name and which contract is

responsible for resolving it.[11] This singular focus greatly reduces the contract's complexity and attack surface, allowing it to serve as a permanent and reliable foundation for the entire ecosystem.

Ultimate control over the ENS protocol is decentralized through a hierarchical ownership structure. The ENS Registry contract itself is owned by another contract known as the ENS Root. The ENS Root, in turn, is owned by the ENS DAO Wallet, a multisig controlled by the ENS Decentralized Autonomous Organization (DAO).[10] This ensures that any administrative changes to the protocol's core, such as upgrading the Registry or managing TLDs, must pass through the DAO's formal governance process, placing ultimate authority in the hands of

$ENS token holders.

The Registry's public interface provides a set of functions for reading and modifying these core records. Read-only functions like owner(bytes32 node), resolver(bytes32 node), and ttl(bytes32 node) allow any external party to query the state of a name. State-changing functions, such as setOwner(bytes32 node, address owner), setResolver(bytes32 node, address resolver), and the crucial setSubnodeOwner(bytes32 node, bytes32 label, address owner) for creating subdomains, are permissioned and can only be called by the current owner of the respective node.[10]

1.2 Registrars: Gatekeepers of the Namespace

While the Registry acts as the ledger, Registrars are the smart contracts that implement the specific rules and business logic for allocating names within a particular namespace.[6] A registrar is simply a smart contract that owns a Top-Level Domain (TLD) node in the Registry (e.g., the

node for .eth) and is therefore authorized to create subdomains under it (e.g., vitalik.eth) by calling setSubnodeOwner on the Registry.[2] This model allows for different allocation mechanisms to coexist within ENS.

The.eth Registrar: A Dual-Contract System

The most prominent registrar is the one governing the native .eth TLD. To enhance security and upgradability, its architecture is strategically split into two separate contracts [13]:

1. **BaseRegistrar:** This contract serves as the ownership layer. It is a robust implementation of the ERC-721 non-fungible token (NFT) standard, where each .eth second-level domain (2LD) is represented as a unique NFT.[5] The
   tokenId for each NFT is the uint256 representation of the name's labelhash. The BaseRegistrar is responsible for managing the ownership of these NFTs, including transfers (safeTransferFrom) and approvals (approve). Its logic is minimal and focused solely on ownership, providing users with a strong and persistent guarantee that they own their name as long as the contract is active.[13]
2. **ETHRegistrarController:** This contract acts as the logic layer, managing the user-facing processes of registration and renewal.[13] It contains the business logic for pricing, interacting with a pricing oracle to convert USD-denominated fees into ETH, and implementing the
   commit-reveal scheme to prevent front-running during registration. When a user registers a name through the controller, it interacts with the BaseRegistrar to mint the new NFT to the user. This separation allows the ENS DAO to upgrade the registration logic (e.g., change pricing models or add new features) by deploying a new controller and pointing the BaseRegistrar to it, all without affecting the ownership records of existing names.[13]

The DNS Registrar

To facilitate the integration of the traditional web, ENS employs a specialized DNS Registrar. This contract is responsible for allowing owners of DNS names (e.g., .com, .org, .xyz) to claim corresponding ownership within ENS.[6] It uses on-chain verification of DNSSEC signatures to cryptographically prove that the user making the claim is the legitimate owner of the DNS name, thus creating a secure bridge between Web2 and Web3 identities.[14]

The Reverse Registrar

The Reverse Registrar is another specialized contract that governs the special-purpose .addr.reverse TLD.[10] Its purpose is to manage reverse resolution, the process of mapping an Ethereum address back to a name. Any user can interact with this registrar to claim the node corresponding to their address (e.g.,

[their_address].addr.reverse) and set a name for it, which is a critical component for

establishing a "Primary Name".[17]

1.3 Resolvers: The Data Translation Layer

If the Registry answers "who owns a name?" and Registrars answer "how are names acquired?", then Resolvers answer the question "what does a name do?".[6] A resolver is a smart contract that a name's owner points to in the Registry. Its purpose is to hold the actual records associated with a name and implement the logic for returning that data in response to queries.[5] This architectural choice to separate the data from the ownership registry is the key to ENS's remarkable extensibility. New types of records and resolution logic can be introduced simply by deploying new resolver contracts, without any changes to the core ENS Registry.[11]

The Public Resolver

For most users, the default resolver is the Public Resolver, a general-purpose, "swiss army knife" contract developed and maintained by the ENS team.[18] It is designed to support a wide array of standardized record types, each defined by a specific EIP or ENSIP. This allows a single ENS name to serve as a multi-faceted digital identity. Key features supported by the Public Resolver include:

- **Address Records:** addr(bytes32 node) for the primary Ethereum address (EIP-137) and addr(bytes32 node, uint256 coinType) for multi-coin support, allowing a name to resolve to Bitcoin, Litecoin, and other cryptocurrency addresses (EIP-2304).[19]
- **Text Records:** text(bytes32 node, string key) for storing arbitrary key-value metadata (EIP-634). This is commonly used for profile information like an avatar (often an NFT link), a description, social media handles (com.twitter, com.github), or a website url.[19]
- **Content Hash:** contenthash(bytes32 node) for pointing to decentralized websites hosted on systems like IPFS or Swarm (EIP-1577).[19]
- **Reverse Resolution:** name(bytes32 node) for mapping a reverse-lookup node back to a human-readable name (EIP-181).[19]
- **Contract ABI:** ABI(bytes32 node, uint256 contentTypes) for storing the Application Binary Interface of a smart contract, making it easier for applications to interact with it (EIP-205).[19]

Custom Resolvers

While the Public Resolver covers most common use cases, the true power of the resolver model lies in its permissionless nature. Any user or developer can write and deploy their own custom resolver contract and point their ENS name to it.[18] This opens up a vast design space for advanced functionalities. For example, a custom resolver could be designed to:

- Dynamically generate resolution data based on other on-chain states.
- Implement access control, returning different data based on the querying address.
- Serve as a gateway for off-chain or L2 data retrieval using the CCIP-Read protocol.

This architectural triad—a stable Registry, rule-based Registrars, and extensible Resolvers—creates a system that is both robust and highly adaptable. The core protocol remains secure and unchanging, while the functionality of ENS can be perpetually expanded by the community through the creation of new registrars and resolvers. This permissionless extensibility is not just a feature but the fundamental design philosophy that ensures ENS can evolve to meet the future needs of the decentralized web.

Table 1.1: Core ENS Contract Deployments (Ethereum Mainnet)

For developers interacting with the protocol directly, having the canonical contract addresses is essential. The following table provides a quick-reference guide to the primary ENS smart contracts deployed on the Ethereum Mainnet.

Contract Name	Address
ENS Registry	0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e
Base Registrar (.eth)	0x57f1887a8BF19b14fC0dF6Fd9B2acc9Af147eA85
ETH Registrar Controller	0x59E16fcCd424Cc24e280Be16E11Bcd56fb0CE547
DNS Registrar	0xB32cB5677a7C971689228EC835800432B339bA2B

Reverse Registrar	0xa58E81fe9b61B5c3fE2AFD33CF304c454AbFc7Cb
Name Wrapper	0xD4416b13d2b3a9aBae7AcD5D6C2BbDBE25686401
Public Resolver	0xF29100983E058B709F3D539b0c765937B804AC15
Universal Resolver	0xED73a03F19e8D849E44a39252d222c6ad5217E1e

Data sourced from.[20]

Section 2: The ENS Resolution Lifecycle: From Name to Resource

The process of resolving an ENS name is a carefully orchestrated sequence of operations that transforms a user-friendly string into a useful on-chain or off-chain resource. This lifecycle begins with rigorous name processing to ensure canonical representation and concludes with a series of smart contract queries. Understanding this flow is critical for any developer building applications that integrate with ENS, as it reveals the protocol's security considerations and operational mechanics.

2.1 Name Processing: Normalization and Hashing

Before an ENS name can be used in any on-chain interaction, it must be converted into a standardized, fixed-size format. This is a two-step process involving normalization and hashing. It is imperative that libraries and applications perform these steps correctly, as even a minor deviation will result in an incorrect hash and a failed lookup.[21]

ENSIP-15 Normalization

The first and most critical step is normalization. ENS names are not simple ASCII strings; they can contain a wide range of Unicode characters, including emojis. To ensure that different representations of the same conceptual name resolve to the same resource, all names must be canonicalized according to the ENSIP-15 standard, which is based on the UTS-46 algorithm.[21] This process handles several key transformations:

- **Case Folding:** All uppercase characters are converted to lowercase (e.g., MyName.ETH becomes myname.eth).
- **Character Mapping:** Certain characters are mapped to their canonical equivalents.
- **Validation:** The algorithm checks for and disallows illegal characters or sequences, preventing spoofing attacks where visually similar characters (homoglyphs) could be used to impersonate another name.

A name is considered valid only if it can be successfully normalized. Modern libraries like viem and ensjs use the @adraffy/ens-normalize package to perform this function, throwing an error if a name is invalid.[21]

Labelhash and Namehash

Once a name is normalized, it is processed into a 32-byte hash for efficient use within smart contracts. ENS uses two distinct hashing functions:

1. **Labelhash:** This is the keccak256 hash of a single, normalized label from a domain name. For example, in sub.domain.eth, the labelhashes for sub, domain, and eth would be calculated independently. The labelhash is used in specific contexts where only a single part of the name is relevant. A primary example is the .eth BaseRegistrar contract, where the ERC-721 tokenId of a name like example.eth is the uint256 representation of keccak256('example').[6]
2. **Namehash:** ENS does not store human-readable names directly on-chain due to their variable length and cost. Instead, it uses a fixed-length 256-bit cryptographic hash known as a **namehash**. This process ensures that every name has a unique, fixed-size identifier. This is the core algorithm used to produce the unique node identifier for a full domain name that is stored in the ENS Registry. It is a recursive algorithm designed to preserve the hierarchical nature of domain names. The process works from right to left (from the TLD inwards).

The namehash algorithm is a recursive process that breaks a name into its components (labels) and hashes them with the hash of the parent domain.

The standard formula for calculating a namehash is:

`namehash(name) = keccak256(namehash(parent_name) + keccak256(label))`

Let's break this down with the example `vitalik.eth`:

1. **Normalization:** The name is first normalized and lowercased.
2. **Splitting Labels:** The name is split into its labels: `vitalik` and `eth`.
3. **Recursive Hashing:** The algorithm works from right to left (from the top-level domain down).
    - The `namehash` of the root node is simply 32 bytes of zero: `0x0000000000000000000000000000000000000000000000000000000000000000`.
    - **To calculate `namehash('eth')`:**
        - First, calculate the hash of the label: `hash_label_eth = keccak256('eth')`
        - Then, combine it with the parent's namehash (the root node): `namehash('eth') = keccak256(namehash(root) + hash_label_eth)`
    - **To calculate `namehash('vitalik.eth')`:**
        - First, calculate the hash of the label: `hash_label_vitalik = keccak256('vitalik')`
        - Then, combine it with the parent's namehash (`namehash('eth')`): `namehash('vitalik.eth') = keccak256(namehash('eth') + hash_label_vitalik)`

This process results in a unique and deterministic 32-byte hash for any valid domain name, which is the key used for all lookups within the ENS Registry.

This recursive structure means that the namehash of a parent domain is required to calculate the namehash of a child domain, mirroring the hierarchical relationship on-chain.[11]

2.2 Forward Resolution: The Canonical Lookup Process

Forward resolution is the most common ENS operation: the process of taking a human-readable name and resolving it to a resource, such as an Ethereum address.[16] The process is a clear, two-step query that leverages the separation between the Registry and Resolvers.

The end-to-end flow for resolving example.eth to its ETH address is as follows:

1. **Client-Side Preparation:** A dApp or wallet takes the input example.eth. It first normalizes the name and then computes its namehash.
2. **Query the Registry for the Resolver:** The client initiates a read-only call to the ENS Registry contract's resolver(bytes32 node) function, passing the computed namehash of example.eth as the argument.[10]
3. **Registry Response:** The Registry contract looks up the node in its storage and returns the address of the resolver smart contract that the owner of example.eth has designated.
4. **Query the Resolver for the Address:** The client now initiates a second read-only call, this time to the resolver address received in the previous step. It calls the addr(bytes32 node) function on the resolver contract, once again passing the namehash of example.eth.[16]
5. **Resolver Response:** The resolver contract executes its internal logic. For the Public Resolver, this involves looking up the address record associated with that node in its own storage. It then returns the final Ethereum address.

This two-step process is fundamental. The Registry never provides the final data; it only directs the client to the correct resolver. This allows the owner of example.eth to change their resolver at any time to a new contract with different logic or record types, without needing to interact with or modify the core Registry.

2.3 Reverse Resolution and the Primary Name

Reverse resolution is the process of translating a machine-readable address back into a human-readable name, a feature critical for enhancing user experience across the Web3 ecosystem.[16] Instead of displaying a long, inscrutable address like

0xb8c...67d5, an application can display nick.eth, making interfaces more intuitive and transactions less prone to error.[4]

The .addr.reverse Namespace

ENS implements reverse resolution through an elegant architectural choice that reuses the existing forward resolution machinery. A special TLD, .addr.reverse, is reserved for this purpose. The name for a given address is constructed by taking its hexadecimal

representation (lowercase, without the 0x prefix) and appending .addr.reverse.[16] For example, the reverse record for the address

0x...67d5 is stored under the ENS name d567...c2b8.addr.reverse.

To perform a reverse lookup, a client simply executes a standard forward resolution on this specially constructed name. It queries the Registry for the resolver of d567...c2b8.addr.reverse, and then calls the name(bytes32 node) function on that resolver, which returns the associated string.[16]

The Primary Name and the Bi-Directional Link

The concept of a "Primary Name" is the cornerstone of a trustworthy reverse resolution system. A name is considered the primary name for an address only if a **bi-directional link** is established and verified [24]:

1. **Forward Resolution:** The ENS name (e.g., nick.eth) must resolve to the Ethereum address (e.g., 0xb8c...67d5).
2. **Reverse Resolution:** The reverse record for the address (d567...c2b8.addr.reverse) must resolve back to the same ENS name (nick.eth).

This two-way binding is not something that is cryptographically enforced by the core ENS protocol itself. An address owner can use the Reverse Registrar to set their reverse record to point to any name they wish, even one they do not own. The protocol allows this. The integrity of the system, therefore, hinges on a crucial verification step that must be performed by every client-side application.

The documentation is unequivocal on this point: a client **MUST** always follow a reverse lookup with a confirmatory forward lookup. If a client looks up 0xb8c...67d5 and gets back nick.eth, it must then immediately perform a forward lookup on nick.eth. Only if that forward lookup returns the original address, 0xb8c...67d5, should the client display nick.eth. If the check fails, the client should display the raw address to prevent spoofing or misconfiguration.[16]

This design reveals a pragmatic approach to a complex problem. Instead of building intricate on-chain mechanisms to enforce these bi-directional links, the protocol provides simple, powerful primitives and defines a social and technical convention for clients to follow. The security and trustworthiness of the Primary Name system are thus a shared responsibility of the entire ENS ecosystem, upheld by the diligent implementation of this verification standard in wallets and dApps.

Section 3: Advanced Capabilities and Scalability Solutions

While the core architecture of ENS provides a robust and secure foundation on Ethereum Layer 1, the protocol has evolved significantly to address the inherent challenges of scalability, cost, and data flexibility. Through a series of advanced features and standards, ENS has transformed into a hybrid system capable of leveraging off-chain systems and Layer 2 networks. These solutions allow developers to build more complex, cost-effective, and dynamic naming applications, pushing the boundaries of what a decentralized naming system can achieve.

3.1 On-chain vs. Off-chain Resolution: A Comparative Analysis via CCIP-Read (EIP-3668)

The primary bottleneck for any purely on-chain system is the cost and latency of Layer 1 transactions. Storing large amounts of data or executing complex resolution logic on Ethereum Mainnet is often impractical. To solve this, ENS has adopted the **Cross-Chain Interoperability Protocol (CCIP) Read**, specified in EIP-3668, as its standard for securely retrieving data from off-chain sources.[25]

The CCIP-Read Mechanism

CCIP-Read is not a protocol in the traditional networking sense but rather a standardized way for a smart contract to communicate to a client that the requested data is not available on-chain and must be fetched from an external source. The mechanism is triggered when a resolver contract, instead of returning data, reverts with a specific, structured error called OffchainLookup. This error is a signal containing all the information a CCIP-aware client needs to continue the resolution process off-chain [25]:

1. sender: The address of the resolver contract that threw the error.
2. urls: An array of HTTP gateway URLs where the client should send its request.
3. callData: The data payload that the client must include in its request to the gateway. This

typically contains the original resolution query (e.g., the ABI-encoded call for addr(node)).
4. callbackFunction: The function selector on the sender contract that the client must call with the data it receives back from the gateway.
5. extraData: Additional data that the callback function might need to process the gateway's response.

The Resolution Flow

The entire process is designed to be transparent to the end-user. When a client attempts to resolve a CCIP-enabled name, the following sequence occurs [25]:

1. The client calls the resolver contract on L1 as it would for any normal ENS name.
2. The resolver immediately reverts with the OffchainLookup error.
3. The client intercepts and decodes this error. It constructs an HTTP request to one of the provided urls, embedding the sender address and callData.
4. The off-chain gateway receives the request, processes the callData, and fetches the required information from its data source (e.g., a database or an L2 network).
5. The gateway returns the data to the client, typically along with a cryptographic proof or signature.
6. The client takes this returned data and makes a second on-chain call, this time to the callbackFunction on the original resolver, passing in the gateway's response.
7. The resolver's callback function executes its verification logic (e.g., checking a signature or validating a state proof) and, if successful, decodes and returns the final, resolved data to the client.

The Trust Spectrum of CCIP-Read

This mechanism enables a spectrum of trust models, allowing developers to make conscious trade-offs between decentralization, security, and cost:

- **Centralized (Trusted) Model:** This is the most common implementation for large-scale, low-cost subname issuance. Projects like Coinbase (cb.id) and Uniswap (uni.eth) use this model. The off-chain gateway is a traditional web server that queries a private database. The data returned to the client is signed by a private key controlled by the organization. The on-chain resolver's callback function simply verifies that the signature matches a trusted public key stored in the contract.[25] This is highly scalable and efficient but introduces a centralized point of failure and censorship.

- **Trust-Minimized (L2) Model:** This model offers a much higher degree of security and decentralization. Projects like Linea (.linea.eth) and Clave (.clv.eth) use this approach. The gateway queries an L2 network for the requested data and also fetches a Merkle proof of that data's inclusion in the L2's state. Since L2s periodically post their state roots to L1, the L1 resolver's callback function can perform a trustless verification of the Merkle proof against the on-chain state root.[25] This model, especially when paired with solutions like "Unruggable Gateways," significantly reduces trust assumptions, relying only on the security of the L2's proving system.

This hybrid approach demonstrates a pragmatic evolution of the ENS protocol. It acknowledges that a one-size-fits-all, purely on-chain solution cannot meet the diverse needs of the Web3 ecosystem. By providing a standardized framework for off-chain data retrieval, ENS allows developers to choose the optimal point on the spectrum of the blockchain trilemma—decentralization, security, and scalability—that best suits their application's requirements.

3.2 Subdomain Architecture and Use Cases

The hierarchical nature of ENS is one of its most powerful features. The owner of any domain, such as mydao.eth, has the unilateral right to create and manage any number of subdomains under it, such as proposals.mydao.eth or member1.mydao.eth.[1] This capability has given rise to a rich ecosystem of subdomain issuance platforms, which can be implemented using L1, L2, or off-chain architectures.

- **L1 Subnames:** This is the most direct and decentralized method. The owner of the parent name calls the setSubnodeOwner function on the ENS Registry to create a new subdomain and assign its ownership. While this offers the full security guarantees of the Ethereum Mainnet, each creation, transfer, or record update is a separate L1 transaction, making it prohibitively expensive for large-scale use cases.[27]
- **L2 Subnames:** A more scalable approach involves the parent name on L1 pointing to a CCIP-Read resolver. This resolver, in turn, directs queries to a dedicated registrar smart contract deployed on an L2 network. Users can then interact with the L2 registrar to mint, manage, and configure their subdomains at a fraction of the cost and with much lower latency. The resolution remains secure through the trust-minimized CCIP-Read flow described above.[27]
- **Off-chain Subnames:** For applications that require issuing millions of free or very low-cost names, the off-chain model is the most practical solution. The parent name's resolver points to a centralized web server. Subdomains are not on-chain tokens but simply records in a database, managed via a REST API. From a user's perspective, resolution is seamless, but these names do not appear as NFTs in their wallets and their

existence is contingent on the continued operation of the central server.[26]

3.3 Wildcard Resolution (ENSIP-10)

ENSIP-10 introduces a significant enhancement to resolver functionality by specifying an optional resolve() function. Standard resolver functions like addr() only receive the namehash of the name being queried. This is efficient but limiting, as the hash is a one-way function; the resolver cannot reconstruct the original name from it. The resolve() function, however, receives the full, DNS-encoded plaintext name as an argument.[16]

This seemingly small change unlocks the possibility of **wildcard resolution**, where a resolver can implement logic that applies to a pattern of names rather than a single, explicitly defined name. For example, a resolver for mynfts.eth could be programmed to handle any query matching the pattern [id].mynfts.eth. When a user looks up 123.mynfts.eth, the resolve() function would receive the full name, parse out the label "123", and then dynamically query an NFT contract to find the owner of token ID 123, returning that owner's address.[28] This type of dynamic, on-the-fly resolution is impossible with

namehash-based functions.

Wildcard resolution is also a key enabler for many CCIP-Read implementations. A resolver for a parent domain like base.eth can be configured to act as a wildcard for all its subdomains (*.base.eth). When any subdomain is queried, the resolver's resolve() function is triggered, which then initiates the OffchainLookup process to fetch the data from the Base L2 network.[26]

Section 4: Bridging Web2 and Web3: DNS Integration

A core tenet of the ENS philosophy is to extend the functionality of the existing internet infrastructure, not to replace it. This is most evident in its robust support for importing traditional DNS names into the ENS ecosystem. This allows the owners of the billions of existing .com, .org, .dev, and other DNS domains to leverage their established identities within Web3, using them as wallet addresses and decentralized profiles without having to register a new .eth name.[1]

4.1 The Role of DNSSEC

The entire mechanism for DNS integration is built upon the foundation of **Domain Name System Security Extensions (DNSSEC)**.[6] DNSSEC is a suite of specifications that adds a layer of cryptographic security to the DNS protocol. It allows the owner of a domain to sign their DNS records with private keys. These signatures can then be publicly verified against public keys that are themselves signed by a higher authority in the DNS hierarchy, creating a verifiable chain of trust that extends all the way up to the root zone, which is signed by ICANN.[14]

For ENS, DNSSEC provides the indispensable, trustless proof of ownership. By requiring that a DNS name be DNSSEC-enabled, the ENS smart contracts can verify on-chain that a user attempting to claim a name like example.com is, in fact, the legitimate owner, because only the true owner possesses the private keys necessary to create the required signatures.[14]

4.2 DNS Name Import Mechanisms

The process of importing a DNS name into ENS involves the domain owner setting a specific TXT record in their DNS zone file. This record acts as a public declaration of their intent to link their DNS name to a specific Ethereum address. ENS supports two distinct methods for verifying this record and activating the name on-chain.

The TXT Record

The process begins with the user adding a TXT record to their domain's DNS configuration. The record must be set on a specific subdomain, typically _ens, or on the root domain itself. The value of this TXT record contains the Ethereum address that will be granted control of the name within ENS. A typical record might look like this [14]:

_ens.example.com. IN TXT "a=0x1234...cdef"

This record publicly states that the owner of example.com wishes to associate it with the address 0x1234...cdef.

Method 1: On-Chain Proof Submission

The original method for importing a DNS name requires the user to submit a transaction directly to the DNSRegistrar smart contract. This transaction must contain a payload of all the cryptographic proofs (the relevant DNS records and their corresponding DNSSEC signatures) that form the chain of trust from the root down to their specific domain. The DNSRegistrar contract then verifies this entire proof chain on-chain.[14]

While this method is fully trustless and self-contained, it has a significant drawback: it is extremely expensive in terms of gas fees. The computational cost of performing cryptographic signature verification for the entire DNSSEC chain on the EVM can run into millions of gas units, creating a high barrier to entry for many users.[14]

Method 2: Gasless Off-Chain Verification

To overcome the cost limitations of the on-chain method, a more modern and efficient "gasless" approach was introduced, leveraging the CCIP-Read protocol.[14] In this model, the user's on-chain interaction is eliminated entirely.

1. **Set TXT Record:** The user configures a slightly different TXT record, in the format ENS1=<resolver-address>, where <resolver-address> is the address of the resolver they wish to use. For a simple address record, a special ExtendedDNSResolver can be used, and the record would look like: ENS1=<extended-resolver-address> <eth-address>.[14]
2. **Resolution Trigger:** When a client attempts to resolve this DNS name in ENS for the first time, the query hits the DNSRegistrar, which acts as a wildcard resolver for DNS TLDs. Seeing no on-chain record for the name, it triggers an OffchainLookup error.
3. **Off-Chain Verification:** The client's CCIP-Read gateway fetches the DNS name's TXT record and the associated DNSSEC proofs directly from DNS servers off-chain.
4. **On-Chain Oracle:** The gateway then submits this proof to an on-chain DNSSECOracle contract as part of the resolution callback. The oracle verifies the proof, and if it is valid, the resolution completes successfully.

With this method, the user pays no gas to "import" or "claim" their name. The cost of verification is shifted to the resolution step and is handled by the infrastructure of the client or gateway performing the query. This dramatically lowers the barrier to entry and has made it feasible for any DNSSEC-enabled domain owner to seamlessly use their name in the ENS ecosystem.[14]

TLD Integration

Beyond individual name imports, ENS also facilitates the integration of entire DNS TLDs. Registry operators of TLDs like .locker or .box can engage with the ENS DAO to have ownership of their corresponding TLD node in the ENS Registry transferred to them. This allows them to deploy their own custom registrar and resolver logic, creating a seamless bridge for all users of their TLD and fostering deeper integration between the Web2 and Web3 naming systems.[29]

Section 5: Protocol Economics and Governance

The long-term viability and decentralized direction of the Ethereum Name Service are underpinned by a carefully designed economic model and a robust community-led governance framework. The protocol's economics are structured to curate the digital commons of the .eth namespace, preventing speculative hoarding while generating sustainable revenue. This revenue, in turn, is managed by the ENS DAO, a decentralized body of token holders who collectively steer the future of the protocol.

5.1 Economic Model: Curating a Digital Commons

ENS generates revenue from two primary sources related to the registration of .eth names: annual fees and temporary premium auctions for expired names. The primary purpose of these fees is not profit maximization, but to serve as an incentive mechanism that ensures the namespace remains a useful and accessible public good.[31]

Length-Based Pricing

The annual fee for registering and renewing a .eth name is determined by the length of the name, priced in USD but paid in ETH. This tiered structure is a deliberate economic design

choice to reflect the relative scarcity and desirability of shorter names.[7]

The pricing is as follows:

- **5+ character names:** $5 USD per year
- **4 character names:** $160 USD per year
- **3 character names:** $640 USD per year

By making shorter, more memorable names significantly more expensive, the protocol disincentivizes squatting—the practice of registering valuable names with no intention of using them, only to sell them later at a high markup. This ensures that these scarce digital assets are more likely to be acquired by individuals or entities who derive genuine utility from them and are willing to pay a premium for their use.[13]

Table 5.1: .eth Name Registration Fee Structure

Name Length	Example	Annual Fee (USD)
5+ Characters	longname.eth	$5
4 Characters	four.eth	$160
3 Characters	tri.eth	$640

Data sourced from.[7]

Temporary Premium Auction

When a .eth name expires, it enters a 90-day grace period during which only the previous owner can renew it.[6] If the name is not renewed within this period, it becomes available for public registration. However, to prevent a frantic race where bots with low-latency connections ("snipers") have an unfair advantage, the name first enters a 21-day

Temporary Premium Auction.[13]

This auction functions as a Dutch auction with a decaying premium. On the first day, the name

is available for registration at its standard annual fee plus a premium of approximately $100 million USD. This premium then decreases exponentially over the 21-day period, eventually reaching $0. The first person willing to pay the current premium plus the registration fee can claim the name.[13] This mechanism ensures that a valuable, newly available name is allocated fairly to the party who values it the most at that moment, as determined by the market, rather than the fastest bot.[13]

Fee Destination and Purpose

All revenue generated from both standard registration fees and temporary premiums is sent to the ETHRegistrarController smart contract. From there, the funds are periodically withdrawn to the ENS DAO Treasury (wallet.ensdao.eth).[13] According to the ENS DAO Constitution, these funds are to be used first and foremost to ensure the long-term viability and continued development of the ENS protocol. Any funds not reasonably required for this primary goal may be used to fund other public goods within the Web3 ecosystem, as decided by ENS governance.[13]

5.2 The ENS DAO: Decentralized Governance in Action

The ENS protocol is governed by the ENS DAO, a decentralized autonomous organization that was established in November 2021 with the launch of the $ENS governance token.[3] The DAO is the ultimate authority over the protocol, responsible for managing the treasury, upgrading core contracts, and setting protocol parameters like pricing.

The $ENS Token

The $ENS token is an ERC-20 token whose sole utility is participation in the governance of the protocol.[3] It does not confer any rights to protocol revenue. To participate, token holders must delegate their voting power, either to their own address or to another community member who acts as a delegate. The weight of a delegate's vote on any proposal is proportional to the total amount of

$ENS delegated to their address.[37] This delegation mechanism encourages active

participation and allows for the emergence of informed representatives who can vote on behalf of passive token holders.

The Governance Process

The ENS DAO operates a structured, multi-stage governance process designed to foster discussion, build consensus, and ensure transparent decision-making. The primary venues for governance are the ENS Discourse forum for discussions, Snapshot for off-chain signaling votes, and on-chain voting portals like Tally and Agora for binding executable proposals.[36]

The typical lifecycle of a proposal is as follows [37]:

1. **Temperature Check:** An informal poll on the Discourse forum to gauge initial community sentiment on an idea.
2. **Draft Proposal:** A formal proposal is drafted and submitted as a pull request to the governance documentation repository, where it is debated and refined by the community.
3. **Active Proposal:** Once the draft is finalized, it moves to a formal vote. For social proposals and constitutional amendments, this is typically a 5-day vote on Snapshot. For on-chain actions, an executable proposal must be submitted.

Proposal Types

There are three main categories of proposals that can be put to the DAO [37]:

1. **Executable Proposal:** This is a proposal for a series of on-chain actions to be executed by the DAO's governance contracts. Examples include transferring funds from the treasury, upgrading a core protocol contract, or changing a parameter in the .eth registrar. To submit an executable proposal, a delegate must have at least 100,000 $ENS (0.1% of the total supply) delegated to them. These proposals are subject to a 7-day on-chain voting period followed by a minimum 2-day timelock before execution, providing a final window for security review.
2. **Social Proposal:** This is a proposal that seeks the DAO's agreement on an issue that cannot be enforced by code. Examples include approving a working group's budget, changing governance rules, or making a formal statement on behalf of the DAO.
3. **Constitutional Amendment:** This is a special type of social proposal that seeks to amend the ENS DAO Constitution. Due to its foundational nature, it has a higher approval threshold, requiring a two-thirds majority to pass.

Through this framework, the ENS DAO manages a substantial treasury, funded by protocol revenue, to support the ongoing growth of the ecosystem. It allocates funds to various working groups (e.g., Ecosystem, Public Goods, Meta-Governance) and core development entities like ENS Labs, ensuring that the protocol remains a vibrant and evolving public good, guided by its community of stakeholders.[34]

Section 6: Practical Implementation and Developer Guide

This section provides actionable guidance and code examples for blockchain engineers seeking to interact with the ENS protocol programmatically. The examples will focus on using modern JavaScript libraries such as viem and ethers.js, covering the most common operations from name registration to record management and data querying.

6.1 Registering a .eth Name Programmatically (Commit-Reveal)

Registering a .eth name is a trustless, on-chain process that utilizes a two-transaction **commit-reveal scheme** to ensure fairness and prevent front-running.[13] Front-running occurs when a malicious actor, such as a miner or a bot monitoring the public transaction mempool, sees a pending registration transaction for a valuable name and submits their own transaction with a higher gas fee to register the name first.

The commit-reveal process mitigates this by separating the registration into two phases [39]:

1. **Commit Phase:** The user generates a secret, random value (a salt) and computes a cryptographic hash (a commitment) of the name they wish to register combined with this secret. They then submit a transaction containing only this commitment hash to the ETHRegistrarController contract. Because the hash is a one-way function, observers cannot determine which name is being targeted.[13]
2. **Reveal Phase:** After a mandatory waiting period (typically 60 seconds) to ensure the commit transaction is securely mined, the user submits a second transaction. This "reveal" transaction contains the actual name and the secret value. The contract hashes these inputs and verifies that the result matches the commitment hash stored in the first transaction. If it matches and the name is available, the registration is completed.[13]

Conceptual Code Walkthrough (using viem)

The following provides a conceptual walkthrough of how to programmatically register a name.

JavaScript

```
import { createWalletClient, http, publicClient, parseEther } from 'viem';
import { mainnet } from 'viem/chains';
import { privateKeyToAccount } from 'viem/accounts';
import {
  ensRegistrarControllerAbi,
  ensRegistrarControllerAddress,
} from './contracts'; // Assume these are defined elsewhere

// 1. Setup Clients and Account
const account = privateKeyToAccount('0x...');
const walletClient = createWalletClient({
  account,
  chain: mainnet,
  transport: http(),
});

// 2. Define Registration Parameters
const name = 'my-awesome-name'; // The name to register, without '.eth'
const owner = account.address;
const duration = 31536000; // 1 year in seconds
const secret = crypto.getRandomValues(new Uint8Array(32)); // Generate a secure random 32-byte secret
const resolverAddress = '0xF29100983E058B709F3D539b0c765937B804AC15'; // Public Resolver address

// 3. Make Commitment (Client-Side)
const commitment = await publicClient.readContract({
  address: ensRegistrarControllerAddress,
  abi: ensRegistrarControllerAbi,
  functionName: 'makeCommitment',
  args: [name, owner, duration, secret, resolverAddress,, false, 0],
```

```
});

// 4. Send Commit Transaction
const commitTxHash = await walletClient.writeContract({
  address: ensRegistrarControllerAddress,
  abi: ensRegistrarControllerAbi,
  functionName: 'commit',
  args: [commitment],
});
await publicClient.waitForTransactionReceipt({ hash: commitTxHash });
console.log('Commit transaction confirmed.');

// 5. Wait for the Minimum Commitment Age
console.log('Waiting for 60 seconds...');
await new Promise(res => setTimeout(res, 60000));

// 6. Check Price and Send Reveal (Register) Transaction
const price = await publicClient.readContract({
  address: ensRegistrarControllerAddress,
  abi: ensRegistrarControllerAbi,
  functionName: 'rentPrice',
  args: [name, duration],
});

const registerTxHash = await walletClient.writeContract({
  address: ensRegistrarControllerAddress,
  abi: ensRegistrarControllerAbi,
  functionName: 'register',
  args: [name, owner, duration, secret, resolverAddress,, false, 0],
  value: price.base + price.premium, // Send the required ETH
});
await publicClient.waitForTransactionReceipt({ hash: registerTxHash });
console.log(`Successfully registered ${name}.eth!`);
```

6.2 Managing ENS Records

Once a name is registered, the owner must configure its records to make it useful. This involves two primary steps: setting a resolver for the name and then calling functions on that resolver to set specific records.

Setting the Resolver

By default, a newly registered name does not have a resolver. The owner must first point the name to a resolver contract, which is typically the mainnet Public Resolver. This is done by calling setResolver on the ENS Registry contract.

Conceptual Code (using ethers.js):

JavaScript

```
import { ethers, namehash } from 'ethers';

// Assume provider and signer are already configured
const ensRegistryAddress = '0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e';
const ensRegistryAbi =;
const publicResolverAddress = '0xF29100983E058B709F3D539b0c765937B804AC15';
const name = 'my-awesome-name.eth';

const registryContract = new ethers.Contract(ensRegistryAddress, ensRegistryAbi, signer);

const tx = await registryContract.setResolver(namehash(name), publicResolverAddress);
await tx.wait();
console.log('Resolver set to Public Resolver.');
```

Setting Address, Text, and Content Hash Records

With the resolver set, the owner can now populate the name's records by calling functions on the resolver contract itself.

Conceptual Code (using viem):

JavaScript

```
import { namehash } from 'viem/ens';
import { publicResolverAbi, publicResolverAddress } from './contracts';

const name = 'my-awesome-name.eth';
const node = namehash(name);

// Set the primary ETH Address Record
const setAddrHash = await walletClient.writeContract({
  address: publicResolverAddress,
  abi: publicResolverAbi,
  functionName: 'setAddr',
  args: [node, owner],
});
await publicClient.waitForTransactionReceipt({ hash: setAddrHash });
console.log('ETH address record set.');

// Set a Text Record (e.g., Twitter handle)
const setTextHash = await walletClient.writeContract({
  address: publicResolverAddress,
  abi: publicResolverAbi,
  functionName: 'setText',
  args:,
});
await publicClient.waitForTransactionReceipt({ hash: setTextHash });
console.log('Twitter text record set.');

// Set a Content Hash Record (for a decentralized website)
// Note: The content hash needs to be properly encoded (e.g., for IPFS)
const contentHash = '0x...'; // Encoded IPFS hash
const setContentHash = await walletClient.writeContract({
  address: publicResolverAddress,
  abi: publicResolverAbi,
  functionName: 'setContenthash',
  args: [node, contentHash],
});
await publicClient.waitForTransactionReceipt({ hash: setContentHash });
console.log('Content hash record set.');
```

6.3 Setting a Primary Name (Reverse Resolution)

Setting a Primary Name establishes the crucial bi-directional link between an address and a name. This is accomplished by interacting with the ReverseRegistrar contract, which controls the .addr.reverse namespace.[24] The modern

ReverseRegistrar provides a convenient setName function that handles the entire process in a single transaction: it claims the reverse node for the caller, sets the default resolver, and sets the name record.[17]

Conceptual Code (using ethers.js):

JavaScript

```
import { ethers } from 'ethers';

// Assume provider and signer are already configured
const reverseRegistrarAddress = '0xa58E81fe9b61B5c3fE2AFD33CF304c454AbFc7Cb';
const reverseRegistrarAbi =;
const primaryName = 'my-awesome-name.eth'; // The name to set as primary

const reverseRegistrarContract = new ethers.Contract(reverseRegistrarAddress,
reverseRegistrarAbi, signer);

// Before calling this, ensure that 'my-awesome-name.eth' resolves to the signer's address.
const tx = await reverseRegistrarContract.setName(primaryName);
await tx.wait();
console.log(`Primary name set to ${primaryName}.`);
```

6.4 Querying ENS Data

While direct contract calls are necessary for writing data, reading data is often simplified by high-level abstractions in client libraries. For more complex data needs, the ENS Subgraph provides a powerful alternative.

Using Client Libraries

Modern libraries provide simple, one-line methods for the most common lookups, automatically handling the multi-step resolution process.

Conceptual Code (using viem):

JavaScript

```
import { normalize } from 'viem/ens';

// Forward resolution: Name -> Address
const address = await publicClient.getEnsAddress({
  name: normalize('vitalik.eth'),
});
console.log(`vitalik.eth resolves to: ${address}`);

// Reverse resolution: Address -> Name
const name = await publicClient.getEnsName({
  address: '0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045',
});
console.log(`Address resolves to: ${name}`);

// Get a text record
const twitterHandle = await publicClient.getEnsText({
  name: normalize('vitalik.eth'),
  key: 'com.twitter',
});
console.log(`vitalik.eth Twitter: ${twitterHandle}`);
```

Using The Graph for Advanced Queries

For queries that are inefficient or impossible with direct contract calls—such as finding all names owned by an account or searching for subdomains—the ENS Subgraph is the ideal tool.[45] It provides an indexed and queryable database of ENS on-chain events, accessible via

a GraphQL API.

Sample GraphQL Query:

GraphQL

```
# Query to find all.eth domains owned by a specific address
query getDomains($owner: String!) {
  domains(where: { owner: $owner }) {
    name
    labelName
    expiryDate
  }
}

# Variables:
# { "owner": "0xd8da6bf26964af9d7eed9e03e53415d37aa96045" }
```

This query would return a list of all names directly owned by the specified address, along with their expiration dates—data that would require iterating through potentially millions of events to retrieve from the chain directly.[45]

Conclusion

The Ethereum Name Service has firmly established itself as more than a simple naming utility; it is a critical and sophisticated protocol for decentralized identity and resource location in the Web3 ecosystem. Its power derives from a meticulously designed architecture that is at once robust, modular, and remarkably extensible. The deliberate separation of the Registry, Registrars, and Resolvers creates a stable core that underpins a dynamic and permissionless environment for innovation. This architectural triad allows the protocol to evolve without compromising its foundational security guarantees.

The protocol's pragmatic approach to the challenges of blockchain scalability is a key theme of its design. Through the adoption of CCIP-Read and the support for a spectrum of subdomain implementations—from fully decentralized L1 names to trust-minimized L2 systems and centralized off-chain databases—ENS provides a flexible framework for

developers. It acknowledges that the path to mass adoption requires a range of solutions, allowing builders to make conscious trade-offs between decentralization, security, and cost that best suit their specific use cases. This makes ENS not a monolithic, on-chain-only system, but a hybrid protocol built for real-world application.

Furthermore, its commitment to complementing, rather than replacing, the existing internet's DNS demonstrates a forward-thinking strategy for bridging the Web2 and Web3 worlds. By leveraging DNSSEC to create a secure import mechanism, ENS opens the door for billions of existing domain owners to seamlessly extend their identities into the decentralized future.

Governed by a robust DAO and sustained by a well-considered economic model, the ENS protocol is poised for continued growth and integration. As the digital landscape moves increasingly towards Layer 2 solutions and cross-chain interoperability, ENS's adaptable architecture ensures it will remain an integral component of the Web3 stack. It provides the human-readable layer essential for simplifying user interactions, unifying digital identity, and ultimately, making the decentralized web accessible to all.

Works cited

1. What is the Ethereum Name Service? | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/learn/protocol
2. What is Ethereum Name Service (ENS)? - GeeksforGeeks, accessed September 13, 2025, https://www.geeksforgeeks.org/solidity/what-is-ethereum-name-service-ens/
3. Ethereum Name Service price today, ENS to USD live price, marketcap and chart | CoinMarketCap, accessed September 13, 2025, https://coinmarketcap.com/currencies/ethereum-name-service/
4. ENS, accessed September 13, 2025, https://ens.domains/
5. What are ENS Domains? Here's What You Should Know - Remote3, accessed September 13, 2025, https://www.remote3.co/blog-post/what-are-ens-domains-here-is-what-you-should-know
6. Terminology | ENS Docs - ENS Documentation, accessed September 13, 2025, https://docs.ens.domains/terminology
7. FAQ | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/faq/
8. What is Ethereum Name Service (ENS) Domain? - 99Bitcoins, accessed September 13, 2025, https://99bitcoins.com/cryptocurrency/ens-review/
9. What Is Ethereum Name Service? (ENS) - Kraken, accessed September 13, 2025, https://www.kraken.com/learn/what-is-ethereum-name-service-ens
10. The Registry | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/registry/ens
11. Know Everything About Ethereum Name Service (ENS) - 101 Blockchains, accessed September 13, 2025, https://101blockchains.com/ethereum-name-service-explained/
12. ENS Documentation - Read the Docs, accessed September 13, 2025,

https://buildmedia.readthedocs.org/media/pdf/ens/latest/ens.pdf
13. ETH Registrar | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/registry/eth
14. DNS Registrar | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/registry/dns
15. DNS on ENS | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/learn/dns
16. Resolution | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/resolution/
17. Reverse Registrars | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/registry/reverse
18. Resolvers Quickstart | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/resolvers/quickstart
19. Public Resolver | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/resolvers/public
20. Deployments | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/learn/deployments/
21. Name Processing | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/contract-api-reference/name-processing
22. Resolution | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/learn/resolution
23. Design Guidelines | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/dapp-developer-guide/front-end-design-guidelines
24. Primary Names | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/web/reverse
25. Offchain / L2 Resolvers | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/resolvers/ccip-read
26. Layer 2 & Offchain Resolution | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/ccip
27. Subdomains | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/web/subdomains
28. ENSIP-10: Wildcard Resolution | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/ensip/10
29. [EP 6.17] [Executable] Transfer .locker TLD to Orange Domains LLC, accessed September 13, 2025, https://docs.ens.domains/dao/proposals/6.17/
30. ENS Labs Unveils Integration with .box Domains: A New Era for Web3 Identities | ENS Blog, accessed September 13, 2025, https://ens.domains/blog/post/ens-integrates-dot-box
31. Fees | support.ens.domains, accessed September 13, 2025, https://support.ens.domains/en/articles/7900605-fees
32. Protocol Revenue - ENS DAO Basics, accessed September 13, 2025, https://basics.ensdao.org/protocol-revenue
33. Panning for gold.eth: Understanding and Analyzing ENS Domain Dropcatching - Nick Nikiforakis, accessed September 13, 2025, https://www.securitee.org/files/ens_domains_imc2024.pdf

34. ENS DAO Contracts, accessed September 13, 2025, https://basics.ensdao.org/ens-dao-contracts
35. Funding Requests - ENS DAO Basics, accessed September 13, 2025, https://basics.ensdao.org/funding-requests
36. Welcome to ENS DAO | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/dao/
37. Governance Process | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/dao/governance/process/
38. ENS | Treasury - Tally.xyz, accessed September 13, 2025, https://www.tally.xyz/gov/ens/treasury
39. Commit Reveal Scheme on Ethereum. Hiding actions and generating random... | by Austin Thomas Griffith | Gitcoin | Medium, accessed September 13, 2025, https://medium.com/gitcoin/commit-reveal-scheme-on-ethereum-25d1d1a25428
40. Commit-Reveal Scheme - Smart Contract Design Patterns, accessed September 13, 2025, https://andrej.hashnode.dev/commit-reveal-scheme
41. ensdomains/ens-contracts: The core contracts of the ENS protocol - GitHub, accessed September 13, 2025, https://github.com/ensdomains/ens-contracts
42. Registering manually via Etherscan - 0.6 eth bounty for guide - General Discussion, accessed September 13, 2025, https://discuss.ens.domains/t/registering-manually-via-etherscan-0-6-eth-bounty-for-guide/10148
43. How to Set as Primary Name | support.ens.domains, accessed September 13, 2025, https://support.ens.domains/en/articles/8684192-how-to-set-as-primary-name
44. Naming Contracts | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/web/naming-contracts
45. Subgraph | ENS Docs, accessed September 13, 2025, https://docs.ens.domains/web/subgraph