[SHARED] SPP3: Goldsky - Indexing Application

Introduction

eRPC and Goldsky have submitted a proposal to help build ENS into the identity layer for agents. That proposal has a narrow scope of required indexing, and we were asked what a more complete indexing proposal (with or without the agentic payments scope) would look like.

This document outlines that “alternative” proposal. One framing note before we get into it below: we are not proposing to replace ENSNode. ENSNode is production-ready, has communicated clearly through SPP2, and is much more embedded in the ENS ecosystem. What we are proposing is a second, architecturally distinct implementation that does things that, to our knowledge, current ENS indexing stacks do not support natively.

Proposal

We are proposing to build a complete, production-grade indexer for all of ENS (v1 and v2) in a single unified schema, with hosted public endpoints and a self-hostable open-source plugin. This would be powered by Streamling, an open-source indexer that presents several major advantages vs. any prior ENS indexer, and one that scales into ENS’s future past v2. Below, we’ll first outline from a technical perspective the scope and architecture we suggest, then highlight from a broader perspective some of the key differences in our approach.

Why now

ENS v2 is launching on Ethereum in the coming weeks, with a foundational difference in data architecture: events are no longer emitted from a fixed set of known contracts, but rather following a dynamic “factory pattern” of `UserRegistry` contracts. “Factory contract indexing” is a well supported pattern by subgraphs, Ponder, and most indexing frameworks, but ENS v2 is unique in that these factories can nest - a `UserRegistry` contract can emit its own `SubregistryUpdated` events - and as we understand - this can theoretically “nest” infinitely. The result is an unbounded, self-expanding contract tree with no compile-time or startup-time enumeration possible, which the current indexing frameworks (subgraphs via `graph-node` and ENSNode via `ponder` are not capable of natively managing.

Both frameworks do support factory patterns, but require the root contract to be known. But with the “recursivity” possible with ENS v2, discovered registries can themselves emit `SubregistryUpdated`, pointing to further registries. Ponder's factory feature requires a fixed factory address in the config. Typical indexing frameworks can manage this by removing the address constraint entirely and watching all contracts on chain for the event selector which works for discovery but creates a provenance gap - there is no check that the emitting

contract was itself a previously-validated registry, this would allow any contract that emits the right event signature to inject arbitrary addresses into the watched set.

Ponder's maintenance trajectory also introduces uncertainty. The Ponder team joined Monad Foundation in February 2026, and the last commit to the repository was March 17. While the project remains usable today, ENS may ultimately benefit from an indexing implementation with an actively maintained core framework. And beyond the continuity risk, we see it as an opportunity to deeply and holistically upgrade ENS's infra stack: aligned with the agentic commerce vision we'd outlined in our previous proposal, but also on it's own.

Scope

On the v1 side:

- Relevant events across ENSRegistry, BaseRegistrar, NameWrapper, ETHRegistrarController, PublicResolver and historical resolver contracts, and ReverseRegistrar

On the v2 side:

- Relevant events across known RootRegistry, ETHRegistry, and ETHRegistrar contracts
- UserRegistry contracts, dynamically discovered via validated `SubregistryUpdated` events
- PermissionedResolver contracts, dynamically discovered via validated `ResolverUpdated` events
- Registry lifecycle and token-mapping events, including `TokenResource` and `TokenRegenerated`
- ERC-1155 transfer events
- EAC role-change events, including `EACRolesChanged`
- BatchRegistrar premigration batch-reservation processing

API surfaces exposed:

- GraphQL endpoint
- REST APIs for name resolution, primary name lookup, and status queries
- Event streams for realtime downstream consumers

- Archival data stored on R2 for cost-efficient backfills and large-scale analytics

Multi-chain support (to be discussed): Ethereum mainnet (v1 + v2), Base (Basenames), Linea, Optimism, Arbitrum, Scroll, and extensibility to future EVM networks as needed.

Self-hostable: Docker Compose stack, Kubernetes Helm chart, MIT-licensed ENS plugin against a Streamling runtime.

New capabilities

Streamling is Goldsky's streaming data pipeline engine. It is the same infrastructure we use to run 3,000+ concurrent pipelines across 150+ chains serving 10,000+ concurrent consumer clients, including production workloads for Polymarket, Ethena, USDT0, Stripe, and many more. This is not a new product being built for this proposal; it is existing production infrastructure being extended with an ENS-specific plugin. Below, we outline some of the specific new capabilities and other benefits this unlocks for the ENS ecosystem's infrastructure layer.

Dynamic discovery: Streamling's dynamic tables allow for continuous read and write to a lookup table in the indexing workflow, with no requirement for the monitored contracts to be known at startup. This means it natively supports the ENS subname layer with no work required at all on the core framework layer, while both `ponder` and `graph-node` would require significant effort.

REST APIs: Many existing ENS indexing deployments are Postgres-centered, and exposing additional API surfaces often requires separate ETL or serving layers. This multi-module architecture increases costs and complexities. However, Streamling supports writing to many different types of syncs, potentially from a single pipeline if desired, with unified checkpointing. This means that enabling REST APIs is trivial; just requires replicating the data feeding the GraphQL API to another database, with another API server.

Semantic and hybrid search: Shared but distinct benefit of this `multi-sink` feature with Streamling allows for it to write to vector databases like Turbopuffer, Lance, Chroma, and more. This means that unique, net-new functionality like autocomplete, fuzzy search, and AI-friendly semantic search become much more feasible than they are today. This is expanded on in greater detail in our original proposal.

Ecosystem resilience and performance: ENSNode is the Ponder-and-Typescript implementation. The subgraph, which is also Typescript-based, also has staying power with the community. Goldsky's streamling-and-Rust implementation adds a novel new architecture and infrastructure to the ecosystem. It's critical to note that we are not claiming a head start on ENS protocol knowledge, or necessarily even parity: The ENSNode and Graph teams have delivered solutions driving hundreds of millions of queries with good performance, and have some v2 domain-specific implementation depth. We are claiming that Streamling's architecture makes it the right framework for ENS: one that is flexible for v2's needs, offers a new and flexible architecture, and delivers greater performance as well: we've been working on adding our results to the open blockchain indexer benchmark and the initial results have Streamling as a top contender in most categories.

Performance and efficiency

On performance and efficiency, indexing the entirety of Solana in real time takes under 5 vCPUs and 4GB of RAM to transform and emit data into an OLAP database, powering another public resource at crypto.clickhouse.com, writing 20,000 events per second. A typical EVM indexing workload takes under 1 CPU, making it easily runnable locally. The full stack requires a database to run, but that is easily extractable to the cloud, so users have a clear path to full-stack ownership.

Our hosted offering is also the fastest indexer according to the Open Blockchain Indexer Benchmark:

Case	Sentio	Envio HyperIndex	Ponder	Subsquid	Subgraph (Hosted)	Sentio Subgraph	Goldsky
case_1_lbtc_event_only	11.02 min	6.94 min	34.80 min	40.94 min	188.79 min	14.90 min	**1.38 min**
case_2_lbtc_full	7.78 min	8.54 min	64.86 min	46.85 min	66.41 min	29.23 min	**2.59 min**
case_3_ethereum_block	2.51 min	N/A	9.63 min	0.25 min ++	50.58 min	2.67 min	**0.33 min (19.7s)**
case_4_on_transaction	22.12 min	N/A	Timeout§	**1.25 min**	N/A	N/A	3.76 min
case_5_on_trace	2.54 min	N/A	74.71 min	7.42 min	17.81 min	2.17 min	**0.75 min (45.0s)**
case_6_template	14.36 min	1.92 min	6.44 min	5.34 min	16.83 min	4.26 min	**0.25 min (15.2s)**

> Notes on benchmark results: two cells warrant explanation:
>
> On case_3, the Subsquid figure (0.25 min) reflects only 13% completeness, so Streamling is the fastest indexer to fully complete that case.
>
> On case_4, the reported Subsquid figure appears to transpose Envio HyperIndex's result from the same case; we've flagged this with the benchmark maintainers and will update once confirmed.
>
> Setting those two aside, Streamling is the fastest complete indexer across every case in the suite.

Our addition to the benchmark is in this branch, currently in the process of being officially included, with the full results here. Self-hosted speeds will vary based on RPC performance.

Milestones

Week 1+2: Core technical validation

Work: v1 event routing, ABI decoder, Postgres sink schema, Docker packaging. ENS v2 dynamic discovery and TokenResource reconciliation. Dynamic table expansion, same-block race condition handling, burn+mint/TokenRegenerated correlation, `registry` and `token_resource` schema.

Deliverable: pipeline ingests all v1 events and writes to any sink. Pipeline also correctly indexes ENS v2 on Sepolia with dynamic registry expansion verified.

Months 1–3: Build phase

Work: Name construction, migration detection, resolver tracking, API server, migration compatibility verification.

Deliverable: full unified API on Sepolia and/or Mainnet if available, GraphQL and REST endpoints live.

Months 4–16: Maintenance and operation

Work and deliverables:

- Hosted public endpoint with SLA commitment (99.5% uptime, with defined incident response)
- Mainnet launch support: BatchRegistrar burst processing, post-launch debugging, any ENS contract ABI updates handled within 72 hours of notice

- Multi-chain deployment (Base, Linea, Optimism, Arbitrum, Scroll)
- Open-source release of the `@goldsky/ens-plugin` crate with documentation

Success Metrics + Quarterly Reporting

Many of the same metrics we outlined in our original proposal still hold true, just for a different scope of APIs! This is also an area where we're open to setting goals / KRs together with the committee rather than setting the bar for success for ourselves.

Ask

We request $450,000 over a 3-month build plus 12-month maintenance period (15 months total), across three core buckets:

- **Engineering (75%):** FTEs building and operating the indexing stack. The primary deliverable is open-source code and a production-grade hosted indexer.
  - Streamling ENS plugin: v1+v2 event handlers, dynamic UserRegistry discovery with provenance validation, TokenRegenerated reconciliation, migration correlation
  - API server: GraphQL, REST endpoints
  - Multi-chain deployment, operational runbooks, ABI update response
- **Infrastructure (20%):** Multi-chain Postgres hosting, Ethereum RPC access, CDN for the hosted public endpoint. Goldsky's existing streaming infrastructure and managed hosting are already in production; incremental costs for an ENS-specific deployment are substantially lower than a greenfield build. Goldsky also operates its own first-party RPC service called Edge, and this also allows for some operational efficiency.
- **Documentation and ecosystem support (5%):** Schema compatibility CI maintenance, developer documentation, integration support for teams building on the hosted endpoint.

Combinations of scopes and proposals

The above has all been discussing a standalone "unified ENS indexing" scope of work. Below, we outline this in the context of our other proposal, and summarize 3 distinct "scopes":

Option A: Agentic Infrastructure only

Budget ask: $300,000

This is our original proposal, submitted separately. We are including it here for completeness. The scope covers:

- a semantic search layer over ENS name data (pgvector + hybrid BM25)
- an eRPC ENS plugin hosted at `ens.erpc.cloud`
- pay-per-query infrastructure that makes ENS names queryable for AI agents

Option A is built on top of existing ENS indexing data. In its standalone form, that means a dependency on ENSNode's public endpoint as a partial data source. The full technical specification, milestones, and team information for Option A are in our original submission. We have not changed that scope or that ask. Following the call, we've removed the $50K developer marketing allocation from this scope, as that function appears to be well covered already and isn't in the intended scope for SPP3. We'll also take the chance here to illustrate some of the queries that this enables:

- "Find ENS names semantically similar to `uniswap.eth`": vector embedding of name label strings enables finding brand-proximate names, typosquatting candidates, and acquisition targets without exact-match or LIKE queries
- "Names expiring in the next 90 days where the label is semantically close to active DeFi protocols": hybrid query combining similarity with structured filter on `expiry_date`
- "Wallet addresses whose ENS `description` text records describe themselves as builders or developers": semantic search over embedded resolver text records

Option B: Unified ENS Indexing

Budget ask: $450,000

This is the scope in the document above. A complete, production-grade indexer for all of ENS in a single unified schema, with a hosted public endpoint and a self-hostable open-source plugin.

Option C: Full vision: Indexing + Agentic

Budget ask: $600,000

Option C combines Option B and Option A into a single coherent platform.

The critical dependency in Option A's standalone form is that the semantic search and agentic layers depend on ENSNode's public endpoint as their data source. Option C eliminates that dependency. The indexing pipeline that powers the GraphQL and REST API is the same pipeline that writes vector database embeddings and event streams. There is no secondary data source, no separate checkpointing, no sync lag between what the resolver API knows and what the semantic search index knows.

Appendix:Q&A

▼ How does this proposal map to the SPP3 criteria?

- **Prior delivery history:** We left the ecosystem grants in the previous proposal! In the `Record` section of the document
- **Scope clarity:** Clear deliverables (self-hostable framework, REST and GraphQL APIs, archival data access in the indexing scope, and then the discovery + payments layer in the agentic scope. Both with a build phase and a maintenance phase.
- **Milestone structure:** Short-term milestones with named and independently verifiable outputs.
- **Adoption, revenue, utility:** Completely net-new ENS revenue created from agent-driven registrations and renewals. Teams building with eRPC (700+ stars, 100 forks, contributions from teams like Circle) get agent-ready ENS search by default. New utility is an agent-first query and payments engine which does not exist today.

▼ Are the timelines flexible?

Yes - the above represents our default project pace for a scope like this with no external factors, but we understand ENS v2 mainnet is approaching. We'd love to discuss and understand the timelines, as we believe that a full indexing scope is mission-critical before mainnet, ideally with sufficient maturity even during testnet. If selected, aligning on the right timeline and expectations would be our top priority.

ENS v2 Complexities

We want to be specific about what makes ENS v2 indexing hard, because the committee should be able to evaluate whether we understand the problem we are proposing to solve. Based on our readthrough of the documents and initial architecture (not shared in this document, though we can share if it would be helpful), this represents our best understanding. We might not have all the facts, and will be sure to collaborate with ENS Labs + ecosystem to ensure what we build accurately captures both the technical specifics and the ecosystem user needs in our solution.

Unbounded registries

We already talk about this above to some degree, but expand on it here. Registry discovery starts from the known RootRegistry and ETHRegistry addresses. A complete indexer also watches known registrar contracts such as ETHRegistrar. Every `SubregistryUpdated(tokenId, IRegistry subregistry, sender)` event adds a new `UserRegistry` to the set that must be indexed. That registry can emit its own `SubregistryUpdated` events. There is no compile-time enumeration of all contracts that will exist. The same pattern applies to resolvers, and requires strict ordering to prevent race conditions. A `LabelRegistered` event can appear in the same block as its parent registry's first `SubregistryUpdated`. The `Registry` table row must exist before the label can be written with a correct fully-qualified name (though there is added complexity here with the many-to-many name heirarchies). Any indexer that processes these out of order produces permanently unnamed label records. We handle this with a ‘fat blocks’ approach to indexing; the entire block structure is processed as a full, in-order unit with all transactions, logs, and traces nested within, ensuring that in-block consistency is guaranteed. The dynamic tables allow for handling recursive factory patterns with no work needed.

`TokenRegenerated`

`tokenId` is not a stable identifier in ENS v2. Effective name-scoped role changes can trigger token regeneration, causing the old token ID to be burned and a new token ID to be minted. The stable identifier is `resource`. This means that `tokenId` can’t be used as a foreign key. We plan to manage this with an explicit

`token_resource` table as an authoritative mapping, and classify events by checking for a matching `TokenRegenerated` before classifying, and generating `resource` from event data without RPC calls.

Aliases

Aliases (`AliasChanged(fromName, toName)` events on resolver contracts) are entirely invisible at the registry layer. The registry won't contain a direct hierarchy entry for alias-derived names, so resolution APIs need to apply resolver-level alias rewriting before final lookup. We are not 100% confident on how we'll want to manage this: our current hypothesis is that the API layer should apply rewrites transparently before record lookup (with some cap on hops) against an `alias` table, populated by `AliasChanged` events on all discovered resolver contracts.

Cost fields

ENS v2 registration can be denominated in any ERC-20. `NameRegistered` and `NameRenewed` events include a `paymentToken: IERC20` field and split the cost into `base` and `premium` components (renewals only the `base`). For now, our plan is to index `payment_token`, `payment_base`, and `payment_premium` as separate fields, and leave normalization to the consumer.

Other

We didn't think they were worth a full discussion in the appendix, but we also noted the v2 domain lifecycle, v1 > v2 migration events, many-to-many name hierarchies, and a few other complexities. If it's worth having a conversation more deeply about this, happy to do so!