CONFIDENTIAL

UNITED KINGDOM INTELLECTUAL PROPERTY OFFICE

PATENT APPLICATION

Applicant

The Bitcoin Corporation Ltd

Inventor

Richard Boase

Date of Preparation

2 March 2026

Title of Invention

Domain Name Tokenization System with Multi-Proof Ownership Verification, Deterministic Pricing, and Revenue Distribution to Token Holders

Field of the Invention

The present invention relates to systems and methods for converting domain name ownership into tradeable blockchain tokens with automated revenue distribution. More particularly, the invention concerns a system that: verifies domain name ownership through a multi-proof bundle operable in serverless computing environments; creates blockchain-inscribed tokens whose symbols are derived directly from the domain name; prices fractional domain interests using a deterministic square-root decay formula; distributes micropayment revenue from domain web traffic to token holders proportionally to their holdings; and provides a progressive decentralisation architecture enabling domain content to migrate from centralised cloud delivery to censorship-resistant on-chain storage without loss of availability.

Background of the Invention

Problem Statement

Domain names are among the most valuable digital assets on the Internet, yet they remain primitive in their ownership and monetisation characteristics. Existing approaches to domain ownership and domain-based revenue suffer from several deficiencies:

1. No Domain Fractionalisation – Domain names are indivisible assets. A domain is either owned or not. There is no native mechanism to fractionalise domain ownership into tradeable shares, enabling multiple parties to hold economic interests in a single domain. An investor who believes a domain will appreciate in value cannot acquire a partial interest in it – they must either purchase the entire domain or not participate at all. This all-or-nothing ownership model excludes the vast majority of potential participants from the domain name economy.

2. No Domain Revenue Sharing – When a domain generates revenue through advertising, subscriptions, micropayments, or other monetisation methods, there is no standardised mechanism to distribute that revenue to multiple stakeholders proportionally based on their ownership stake. Existing revenue-sharing arrangements require bespoke legal contracts and manual payment processes. No system provides automated, transparent, on-chain revenue distribution to domain token holders in proportion to their holdings.

3. No Serverless Domain Verification – Traditional domain ownership verification requires DNS lookups via system utilities (dig, nslookup, host) that are unavailable in serverless computing environments such as Vercel Edge Functions, Cloudflare Workers, AWS Lambda, and similar platforms. These platforms do not expose system-level DNS resolution capabilities. No standardised multi-method verification bundle exists that works across all deployment environments, including serverless platforms that lack access to traditional DNS utilities. This forces domain verification services to maintain dedicated server infrastructure solely for DNS resolution, increasing cost and complexity.

4. No Deterministic Domain Pricing – Domain name valuation is subjective and opaque. Domains are priced by negotiation, auction, or algorithmic estimates that vary wildly between providers. When fractional interests in a domain are offered, there is no algorithmic pricing mechanism that provides transparent, mathematically deterministic pricing. No party can independently compute the current price of a fractional domain interest from publicly observable state without relying on a centralised price oracle or order book.

5. No Domain-as-Token-Symbol Convention – Existing token systems use arbitrary ticker symbols (e.g., $BTC, $ETH, $DOGE) that are disconnected from the underlying asset and require a registry to map between the symbol and the asset it represents. There is no convention mapping a domain name directly to its token symbol, creating a human-readable,

self-documenting link between the domain and its token. This absence forces reliance on external registries for symbol-to-asset resolution and creates opportunities for symbol squatting and confusion.

6. No Progressive Decentralisation for Domain Content – Domain content is either centrally hosted (fast, low-latency, but censorable and dependent on a single provider) or fully on-chain (censorship-resistant but slow and expensive to serve). There is no architecture enabling progressive migration from centralised to decentralised delivery, wherein a domain begins with cloud-hosted content for performance and incrementally builds on-chain content backups, with automatic fallback to on-chain storage when cloud infrastructure becomes unavailable. Domain owners are forced to choose between performance and censorship resistance, with no middle path.

Prior Art Limitations

DNS registrars (GoDaddy, Namecheap, Google Domains, Porkbun) provide domain registration and management but do not tokenize domains, do not enable fractional ownership, and do not distribute revenue to multiple stakeholders. Domain parking services generate advertising revenue from unused domains but do not fractionalise ownership or distribute revenue to token holders.

ENS (Ethereum Name Service) and Handshake provide blockchain-based naming systems but are alternative namespace systems – they create new top-level domains (e.g., .eth, custom Handshake TLDs) rather than tokenizing existing ICANN domains. A .eth name is not the same asset as a .com name; the two systems are orthogonal. Neither ENS nor Handshake tokenizes the ownership of existing ICANN domains such as .com, .org, or country-code TLDs.

Unstoppable Domains provides blockchain-based domain names but operates alternative TLDs (.crypto, .nft, .x) that are not part of the ICANN domain name system. These domains require custom browser extensions or proxy resolvers to function and are not resolvable by standard DNS infrastructure. Unstoppable Domains does not tokenize existing ICANN domains. Unstoppable Domains has filed patent applications relating to blockchain-based domain resolution, but these address alternative namespace resolution, not the tokenization of existing ICANN domains with fractional ownership, deterministic pricing, and revenue distribution.

3DNS (US Patent 12,147,978, “Systems and Methods for Domain Name Management Using Blockchain”, granted 2024) – 3DNS provides blockchain-based management of traditional domain names, using smart contracts to facilitate domain registration, transfer, and WHOIS record management on-chain. However, 3DNS focuses on domain registration

management (replacing centralised registrar databases with blockchain state) – it does not fractionalise domain ownership via fungible tokens, does not implement algorithmic pricing for fractional interests, does not distribute micropayment revenue to token holders, and does not provide a multi-proof serverless verification mechanism. 3DNS modernises the registrar; the present invention tokenizes the domain as an investable asset.

Doma Protocol (D3 Global, November 2025) – The Doma Protocol enables traditional domain names to be represented as NFTs on blockchain, facilitating domain trading and custody through smart contracts. However, Doma represents each domain as a single non-fungible token (one NFT = one domain = whole ownership), not as a pool of fungible fractional tokens with deterministic pricing. Doma does not implement multi-proof serverless ownership verification (DNS TXT + HTML meta + well-known file), does not use a domain-as-token-symbol convention, does not apply square-root decay pricing for fractional interests, does not distribute micropayment revenue to fractional token holders, and does not implement progressive decentralisation from cloud to blockchain storage. Doma facilitates domain trading; the present invention fractionalises domain ownership and creates a revenue-distributing economic layer.

D3Serve Labs / Namefi (US Patent Application 18/978,071, filed December 2024) – Namefi describes a blockchain-based method for transferring domain name "beneficiary-ship" via smart contract, bypassing registrar pre-approval while maintaining WHOIS/KYC compliance. The system enables features including fractional ownership, with registrars involved only when DNS resource record changes are needed. However, Namefi focuses on domain transfer mechanics (how ownership moves between parties on-chain) rather than on creating an investable economic layer. Namefi does not implement a deterministic pricing mechanism (square-root decay or otherwise), does not distribute micropayment revenue to token holders, does not define a domain-as-token-symbol convention, and does not provide a multi-proof serverless verification mechanism (DNS TXT + HTML meta + well-known file). Namefi solves domain custody transfer; the present invention creates a domain investment and revenue distribution system.

Existing blockchain timestamping services (e.g., OpenTimestamps, OriginStamp) prove that data existed at a particular time but do not bind a timestamp to domain ownership identity, do not create tradeable tokens, do not implement pricing mechanisms, and do not distribute revenue.

The $402 protocol (The Bitcoin Corporation Ltd) defines URL path tokenization – creating tradeable tokens for content at specific URL paths – but does not claim domain name tokenization (tokenizing the domain itself as an asset), multi-proof verification for domain ownership, the domain-as-token-symbol convention, the specific square-root decay pricing

mechanism for domain tokens, or the micropayment revenue distribution and sovereign bridge architecture that are the subject of the present invention.

No existing system combines: multi-proof domain ownership verification operable in serverless environments; a deterministic mapping from domain name to token symbol; on-chain inscription of domain token metadata as ordinals; algorithmically deterministic pricing for fractional domain interests; automated micropayment revenue distribution to domain token holders; and progressive decentralisation of domain content from cloud to blockchain storage.

Summary of the Invention

The present invention provides a system and method for domain name tokenization ("the DNS-DEX System") comprising:

(a) A multi-proof ownership verification bundle – Domain ownership is verified through one or more of three complementary proof methods: a DNS TXT record queried via DNS-over-HTTPS (enabling verification in serverless environments), an HTML meta tag embedded in the domain's web content, and a standardised well-known file at a predetermined path. The three methods form a fallback chain: DNS TXT is attempted first, then HTML meta tag, then well-known file. Any single successful proof suffices to establish ownership. The DNS-over-HTTPS approach queries public resolvers (Google DNS at `https://dns.google/resolve`, Cloudflare at `https://cloudflare-dns.com/dns-query`) using standard HTTP requests, enabling domain verification to execute in serverless computing environments where system-level DNS utilities are unavailable.

(b) A domain-as-token-symbol convention – The token symbol for a tokenized domain is derived directly from the domain name by prefixing it with the dollar sign: `$<domain.tld>`. For example, the domain `b0ase.com` produces the token symbol `$b0ase.com`; the domain `coca-cola.com` produces `$coca-cola.com`; the domain `amazon.co.uk` produces `$amazon.co.uk`. Symbol uniqueness is guaranteed by the uniqueness of domain names in the DNS – no two domains can have the same name, therefore no two domain tokens can have the same symbol. The dollar prefix signals economic activity, consistent with the $402 protocol's `$address` convention for monetary signalling. This convention extends the $402 URL path tokenization convention upward to the domain level itself.

(c) On-chain inscription of domain token metadata as ordinals – Verified domain tokens are inscribed on the BSV blockchain as 1-satoshi ordinal inscriptions. The inscription contains structured metadata including: the protocol identifier (`$402`), the operation type (`publish`), the content type (`domain_verification`), the domain name, the token supply, pricing

parameters, the x402 fee, the ownership proof reference, and a timestamp. The inscription creates an immutable, blockchain-anchored deed proving which domain was tokenized, by whom (linked to the $401 identity protocol), when, and with what parameters. The transaction ID serves as the canonical reference for the domain token.

(d) Square-root decay deterministic pricing – The price of each fractional domain token is determined by the formula: `price = base_price / sqrt(supply_remaining + 1)`, where `supply_remaining` is the number of tokens not yet distributed. This formula produces a monotonically increasing price curve: when supply_remaining is large (early purchases), the denominator is large and the price is low; as supply_remaining decreases (later purchases), the denominator shrinks and the price increases. The pricing is deterministic – any party can independently compute the current price from publicly observable state. No order book, no price negotiation, and no centralised price oracle are required. Early participants are mathematically guaranteed to pay less than later participants, creating an incentive for early adoption. The system includes batch pricing for acquiring multiple tokens in a single transaction.

(e) Micropayment revenue distribution to token holders – Domain visitors pay micropayments via the HTTP 402 payment protocol when accessing gated content. Revenue is split according to a declared ratio between the domain owner and a distribution pool. Token holders receive proportional dividends from the distribution pool based on their token holdings. Revenue events are recorded on-chain, providing transparent accounting. The x402 payment indexing enables real-time revenue tracking and dividend computation.

(f) Sovereign bridge progressive decentralisation architecture – Domain content is delivered through a three-phase progressive decentralisation model. In Phase 1 (cloud-primary), content is served from cloud infrastructure for standard web performance while domain token metadata is inscribed on-chain for ownership proof. In Phase 2 (hybrid), periodic snapshots of domain content are archived to blockchain-based distributed storage, creating verifiable content backups. In Phase 3 (on-chain fallback), if cloud infrastructure becomes unavailable, protocol resolvers automatically fall back to serving content from blockchain storage. Token holders may vote via a governance mechanism to redirect the domain to alternative infrastructure. This architecture enables domain owners to benefit from cloud speed while progressively building censorship-resistant fallback infrastructure.

Detailed Description of the Invention

1. Multi-Proof Ownership Verification

The system verifies that the party requesting tokenization of a domain name is the legitimate owner or controller of that domain. Verification proceeds through a multi-proof bundle comprising three independent verification methods, any one of which suffices to establish ownership. The methods are attempted in fallback order: DNS TXT record first, HTML meta tag second, well-known file third.

1.1 Verification Code Generation

Upon initiating the tokenization process for a domain, the system generates a unique cryptographic verification code. The code is derived from a combination of the domain name, the requestor's wallet address, and a timestamp-based seed. The code is deterministic for a given set of inputs but unique to each verification attempt, ensuring that a verification code from one domain cannot be replayed against another domain, and that a verification code from one wallet address cannot be replayed by a different wallet address. The code is encoded as a base-36 string of a fixed length (default: 32 characters).

The verification code serves as a cryptographic challenge: the domain owner must demonstrate the ability to publish the code at a location controlled by the domain's infrastructure (DNS records, web server content, or file system), thereby proving control over the domain.

1.1.1 Verification Code Generation Algorithm

The verification code is generated using the following algorithm. The inputs are: the fully-qualified domain name (FQDN), the requestor's BSV wallet address (a Base58Check-encoded string), and a timestamp seed (Unix epoch seconds, truncated to the nearest hour to provide a one-hour validity window).

Pseudocode:

```
function generateVerificationCode(domain, walletAddress, timestampSeed):
    // Step 1: Normalize inputs
    normalizedDomain = lowercase(trim(domain))
    normalizedAddress = trim(walletAddress)
    // Truncate timestamp to nearest hour (3600-second boundary)
    hourSeed = floor(timestampSeed / 3600) * 3600

    // Step 2: Concatenate inputs with a fixed delimiter
    // The pipe character "|" is chosen because it cannot appear
    // in domain names or Base58Check addresses
    payload = normalizedDomain + "|" + normalizedAddress + "|" +
toString(hourSeed)

    // Step 3: Compute SHA-256 hash of the payload
    hashBytes = SHA256(UTF8_ENCODE(payload))
    // hashBytes is a 32-byte (256-bit) array

    // Step 4: Encode as base-36 (digits 0-9 and letters a-z)
    // Convert the 32-byte hash to a big integer, then repeatedly
    // divide by 36, collecting remainders as base-36 digits
    bigInt = bytesToBigInteger(hashBytes)
    base36Chars = "0123456789abcdefghijklmnopqrstuvwxyz"
    result = ""
    while length(result) < 32:
        remainder = bigInt MOD 36
        result = base36Chars[remainder] + result
        bigInt = floor(bigInt / 36)

    // Step 5: Return the 32-character verification code
    return result
```

Properties of the algorithm:

- **Deterministic**: The same inputs always produce the same code, allowing both the system and the domain owner to independently compute the expected code.
- **Domain-bound**: Changing the domain input produces a completely different code (avalanche property of SHA-256).
- **Wallet-bound**: Changing the wallet address produces a completely different code, preventing code theft across wallet addresses.
- **Time-limited**: The hour-truncated timestamp seed means codes expire after at most one hour, limiting the replay window.
- **Fixed length**: The base-36 encoding is always exactly 32 characters, providing 32 * log2(36) = approximately 165 bits of entropy, which is sufficient to prevent brute-force

guessing.

1.2 DNS TXT Record Verification

The primary verification method requires the domain owner to add a TXT record to the domain's DNS configuration. The TXT record may be placed at any of the following subdomains, checked in order:

- `_dnsdex.<domain>` – dedicated verification subdomain for DNS-DEX
- `_path402.<domain>` – verification subdomain for the $402 protocol
- The root domain itself

The TXT record value must contain the verification code prefixed with one of the accepted verification identifiers: `dnsdex-verify=<code>` or `path402-verify=<code>`.

Critically, the DNS query is performed not via system-level DNS utilities (which are unavailable in serverless environments) but via DNS-over-HTTPS (DoH). The system issues standard HTTPS requests to public DNS resolvers:

- **Primary**: Google Public DNS at `https://dns.google/resolve?name=<domain>&type=TXT`
- **Fallback**: Cloudflare DNS at `https://cloudflare-dns.com/dns-query?name=<domain>&type=TXT`

The response is a JSON document containing the DNS answer section. The system parses TXT records (DNS record type 16) from the answer, strips surrounding quotation marks, and searches for a record matching the expected verification prefix and code.

1.2.1 DNS-over-HTTPS Request Specification

The exact HTTP request issued to the Google Public DNS resolver is as follows:

```
GET /resolve?name=_dnsdex.example.com&type=TXT HTTP/1.1
Host: dns.google
Accept: application/dns-json
User-Agent: DNSDex-Verification/1.0
```

If the Google resolver is unreachable or returns a non-200 status code, the system falls back to the Cloudflare resolver:

```
GET /dns-query?name=_dnsdex.example.com&type=TXT&ct=application/dns-json
HTTP/1.1
Host: cloudflare-dns.com
Accept: application/dns-json
User-Agent: DNSDex-Verification/1.0
```

Both requests use HTTPS (TLS 1.2 or higher). The system enforces a connection timeout of 5,000 milliseconds and a response timeout of 10,000 milliseconds. If both resolvers fail, the DNS TXT verification method is marked as failed and the system proceeds to the next verification method.

1.2.2 DNS-over-HTTPS Response Structure

The expected JSON response from Google Public DNS has the following structure:

```
{
  "Status": 0,
  "TC": false,
  "RD": true,
  "RA": true,
  "AD": false,
  "CD": false,
  "Question": [
    {
      "name": "_dnsdex.example.com.",
      "type": 16
    }
  ],
  "Answer": [
    {
      "name": "_dnsdex.example.com.",
      "type": 16,
      "TTL": 300,
      "data": "\"dnsdex-verify=a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6\""
    }
  ]
}
```

Key fields: - `Status` : 0 indicates NOERROR (successful query). Non-zero values indicate DNS-level errors (e.g., 3 = NXDOMAIN, domain does not exist). - `Answer` : An array of DNS resource records. Each element has `name` , `type` , `TTL` , and `data` fields. - `type` : 16 indicates a TXT record. The system filters the Answer array to include only records with

`type` equal to 16. - `data` : The TXT record value, typically enclosed in double quotation marks by the resolver.

The Cloudflare resolver returns an identical JSON structure when the `ct=application/dns-json` parameter is included.

1.2.3 TXT Record Parsing Algorithm

The system parses the DNS-over-HTTPS response and searches for a matching verification record using the following algorithm:

Pseudocode:

```
function verifyDnsTxt(domain, expectedCode):
    subdomains = ["_dnsdex." + domain, "_path402." + domain, domain]
    resolvers = [
        "https://dns.google/resolve",
        "https://cloudflare-dns.com/dns-query"
    ]

    for each subdomain in subdomains:
        for each resolver in resolvers:
            try:
                url = resolver + "?name=" + subdomain + "&type=TXT"
                response = HTTP_GET(url, headers={
                    "Accept": "application/dns-json",
                    "User-Agent": "DNSDex-Verification/1.0"
                }, timeout=10000)

                if response.status != 200:
                    continue  // try next resolver

                json = parseJSON(response.body)

                if json.Status != 0:
                    continue  // DNS-level error, try next resolver

                if json.Answer is null or empty:
                    continue  // no records found

                for each record in json.Answer:
                    if record.type != 16:
                        continue  // not a TXT record

                    // Strip surrounding quotation marks from data
                    txtValue = record.data
                    txtValue = stripLeadingAndTrailing(txtValue, '"')

                    // Check for verification prefix match
                    // Regex: /^(dnsdex-verify|path402-verify)=(.+)$/
                    match = regex(txtValue, "^(dnsdex-verify|path402-
verify)=(.+)$")
                    if match is not null:
                        extractedCode = trim(match.group(2))
                        if extractedCode == expectedCode:
                            return {
                                verified: true,
                                method: "dns_txt",
                                subdomain: subdomain,
```

```
                                resolver: resolver,
                                rawRecord: txtValue,
                                timestamp: currentTimestamp()
                            }
                        else:
                            // Code found but doesn't match -- helpful
error
                            return {
                                verified: false,
                                method: "dns_txt",
                                error: "CODE_MISMATCH",
                                expected: expectedCode,
                                found: extractedCode
                            }
            catch (networkError):
                continue  // try next resolver

    return { verified: false, method: "dns_txt", error:
"NO_RECORD_FOUND" }
```

The regex pattern `^(dnsdex-verify|path402-verify)=(.+)$` matches verification records with either the `dnsdex-verify` or `path402-verify` prefix. The captured group 2 contains the verification code. The code is trimmed of whitespace before comparison.

This DNS-over-HTTPS approach is the key innovation enabling domain verification in serverless computing environments. Standard `fetch()` or `XMLHttpRequest` calls to HTTPS endpoints are available in all modern computing environments, including Vercel Edge Functions, Cloudflare Workers, AWS Lambda, Deno Deploy, and similar serverless platforms. By performing DNS resolution through HTTP rather than through system-level utilities, the verification process becomes universally deployable without infrastructure dependencies.

The system queries all three subdomain locations in parallel for performance, returning a positive verification result if any location contains a matching record. If a verification record is found but the code does not match, the system returns a specific error indicating a code mismatch, assisting the domain owner in debugging their DNS configuration.

1.3 HTML Meta Tag Verification

The second verification method requires the domain owner to add a meta tag to the HTML content of the domain’s homepage. The meta tag follows the convention:

```
<meta name="dnsdex-verify" content="<verification_code>" />
```

The system fetches the domain's homepage (`https://<domain>`) using a standard HTTP GET request with a `DNSDex-Verification/1.0` user agent string. The system parses the returned HTML using a regular expression to locate a `<meta>` element with `name="dnsdex-verify"` and extracts the `content` attribute value. If the content value matches the expected verification code, ownership is confirmed.

1.3.1 HTML Meta Tag Fetch Specification

The exact HTTP request issued to the domain's homepage is:

```
GET / HTTP/1.1
Host: <domain>
User-Agent: DNSDex-Verification/1.0
Accept: text/html
Accept-Encoding: gzip, deflate
Connection: close
```

The request is issued over HTTPS (`https://<domain>/`). If the domain does not support HTTPS, the system falls back to HTTP (`http://<domain>/`). The system enforces the following timeouts and redirect rules:

- **Connection timeout**: 5,000 milliseconds.
- **Response timeout**: 10,000 milliseconds.
- **Maximum response body size**: 1,048,576 bytes (1 MiB). If the response exceeds this size, the body is truncated. The meta tag is expected to appear within the `<head>` section of the HTML, typically within the first few kilobytes.
- **Maximum redirects**: 5. The system follows HTTP 301, 302, 307, and 308 redirects up to 5 hops. If the redirect chain exceeds 5 hops, the verification method is marked as failed.
- **Final URL recording**: The system records the final URL after all redirects, which is included in the verification proof for auditability.

1.3.2 Meta Tag Extraction Regex

The system extracts the meta tag content using the following regular expression, applied to the response body as a string:

```
/<meta\s+[^>]*name\s*=\s*["']dnsdex-verify["'][^>]*content\s*=\s*["']
([^"']+)["'][^>]*\/?>/i
```

This regex also handles the case where `content` appears before `name` :

```
/<meta\s+[^>]*content\s*=\s*["']([^"']+)["'][^>]*name\s*=\s*["']dnsdex-
verify["'][^>]*\/?>/i
```

Both patterns are attempted. The captured group 1 contains the verification code. The code is trimmed of leading and trailing whitespace before comparison with the expected code.

Pseudocode:

```
function verifyHtmlMeta(domain, expectedCode):
    urls = ["https://" + domain + "/", "http://" + domain + "/"]

    for each baseUrl in urls:
        try:
            response = HTTP_GET(baseUrl, headers={
                "User-Agent": "DNSDex-Verification/1.0",
                "Accept": "text/html"
            }, timeout=10000, maxRedirects=5, maxBodySize=1048576)

            if response.status != 200:
                continue

            body = response.body

            // Pattern 1: name before content
            pattern1 = /<meta\s+[^>]*name\s*=\s*["']dnsdex-verify["']
[^>]*content\s*=\s*["']([^"']+)["'][^>]*\/?>/i
            // Pattern 2: content before name
            pattern2 = /<meta\s+[^>]*content\s*=\s*["']([^"']+)["']
[^>]*name\s*=\s*["']dnsdex-verify["'][^>]*\/?>/i

            match = regex(body, pattern1)
            if match is null:
                match = regex(body, pattern2)

            if match is not null:
                extractedCode = trim(match.group(1))
                if extractedCode == expectedCode:
                    return {
                        verified: true,
                        method: "html_meta",
                        url: response.finalUrl,
                        timestamp: currentTimestamp()
                    }
                else:
                    return {
                        verified: false,
                        method: "html_meta",
                        error: "CODE_MISMATCH",
                        expected: expectedCode,
                        found: extractedCode
                    }
        catch (networkError):
            continue
```

```
    return { verified: false, method: "html_meta", error:
"META_TAG_NOT_FOUND" }
```

This method is advantageous for domain owners who have control over their web server's HTML content but do not have access to their DNS management interface (e.g., because DNS is managed by a third party or because the registrar's DNS interface is complex to navigate).

1.4 Well-Known File Verification

The third verification method requires the domain owner to place a plain-text file at a standardised path:

```
https://<domain>/.well-known/dnsdex-verify.txt
```

This follows the RFC 8615 convention for well-known URIs. The file must contain the verification code as its sole content, with no additional whitespace or characters.

The system fetches the file using a standard HTTP GET request and compares the trimmed file content against the expected verification code. If they match, ownership is confirmed.

1.4.1 Well-Known File Fetch Specification

The exact HTTP request is:

```
GET /.well-known/dnsdex-verify.txt HTTP/1.1
Host: <domain>
User-Agent: DNSDex-Verification/1.0
Accept: text/plain
Connection: close
```

The request is issued over HTTPS. The system enforces:

- **Connection timeout**: 5,000 milliseconds.
- **Response timeout**: 10,000 milliseconds.
- **Maximum response body size**: 4,096 bytes. Verification codes are 32 characters; any response beyond 4 KiB is unexpected and truncated.
- **Maximum redirects**: 3. Well-known files should be served directly; excessive redirects suggest misconfiguration.
- **Expected Content-Type**: `text/plain`. The system accepts any Content-Type but logs a warning if it is not `text/plain`.

Trimming rules: The response body is trimmed as follows before comparison:

1. Remove any UTF-8 BOM (byte order mark: bytes `0xEF 0xBB 0xBF`) from the start of the body.
2. Strip all leading and trailing whitespace characters (spaces, tabs, newlines, carriage returns).
3. If the result contains any newline characters, take only the first line (everything before the first `\n` or `\r\n`).

Pseudocode:

```
function verifyWellKnownFile(domain, expectedCode):
    url = "https://" + domain + "/.well-known/dnsdex-verify.txt"

    try:
        response = HTTP_GET(url, headers={
            "User-Agent": "DNSDex-Verification/1.0",
            "Accept": "text/plain"
        }, timeout=10000, maxRedirects=3, maxBodySize=4096)

        if response.status != 200:
            return { verified: false, method: "well_known_file", error:
"HTTP_" + response.status }

        body = response.body

        // Remove UTF-8 BOM if present
        if body starts with bytes [0xEF, 0xBB, 0xBF]:
            body = body[3:]

        // Trim whitespace
        body = trim(body)

        // Take first line only
        if body contains "\n":
            body = body.split("\n")[0]
        body = trim(body)

        if body == expectedCode:
            return {
                verified: true,
                method: "well_known_file",
                url: url,
                timestamp: currentTimestamp()
            }
        else if length(body) > 0:
            return {
                verified: false,
                method: "well_known_file",
                error: "CODE_MISMATCH",
                expected: expectedCode,
                found: body
            }
        else:
            return { verified: false, method: "well_known_file", error:
"EMPTY_FILE" }
    catch (networkError):
```

```
        return { verified: false, method: "well_known_file", error:
"NETWORK_ERROR" }
```

This method is advantageous for domain owners who can upload files to their web server but cannot modify the HTML source of their homepage (e.g., because the homepage is generated by a content management system that does not support custom meta tags).

1.5 Fallback Chain and Result Aggregation

The three verification methods form a fallback chain. The system attempts DNS TXT verification first. If DNS TXT verification succeeds, the result is returned immediately. If it fails, the system attempts HTML meta tag verification. If that succeeds, the result is returned. If it also fails, the system attempts well-known file verification. If all three methods fail, the system returns a comprehensive error message detailing the failure of each method.

Each verification result includes: the domain name, the verification method that succeeded (or the primary method if all failed), the proof artifact (the matching DNS record, HTML meta tag, or file URL), a timestamp, and any raw records retrieved during the process. This metadata is subsequently referenced in the on-chain inscription, creating a cryptographic chain from DNS ownership to verification proof to blockchain token.

2. Domain-as-Token-Symbol Convention

The system establishes a direct, deterministic mapping from domain names to token symbols:

Convention: The token symbol is `$<domain.tld>` – the dollar sign followed by the exact domain name including the top-level domain.

Examples:

Domain	Token Symbol
b0ase.com	$b0ase.com
coca-cola.com	$coca-cola.com
amazon.co.uk	$amazon.co.uk
example.org	$example.org
wikipedia.org	$wikipedia.org

Properties of the convention:

- **Uniqueness**: The Domain Name System guarantees that no two domain names are identical. Since the token symbol is derived by a deterministic function (dollar-sign

prefixing) from a unique input (the domain name), the output (the token symbol) is also unique. No registry is required to prevent symbol collisions – DNS uniqueness provides this guarantee natively.

- **Human readability**: The token symbol is immediately intelligible to humans. The symbol `$coca-cola.com` unambiguously identifies the underlying asset – the domain `coca-cola.com`. No lookup table, no external registry, and no prior knowledge of a ticker symbol mapping are required.
- **Self-documenting**: The symbol contains the asset's identifier within itself. Unlike arbitrary tickers (where `$BTC` requires knowledge that BTC refers to Bitcoin), the domain-as-token-symbol is self-documenting: the symbol `$b0ase.com` refers to the domain `b0ase.com` by definition.
- **Monetary signalling**: The dollar prefix signals that the symbol represents an economically active asset, consistent with the $402 protocol's convention of using the dollar sign as a monetary signalling character in URLs.
- **Hierarchical extension**: This convention extends the $402 protocol's URL path tokenization upward to the domain level. The $402 protocol tokenizes content at specific URL paths (e.g., `$article-name as a path segment); the present invention tokenizes the domain itself. The two conventions are complementary: $b0ase.com` is the domain token, while `$b0ase.com/$article-name` identifies a specific content token within that domain.

2.1 Domain-to-Symbol Transformation Algorithm

The deterministic transformation from domain name to token symbol follows these exact steps:

Pseudocode:

```
function domainToTokenSymbol(domain):
    // Step 1: Input normalization
    // Convert to lowercase (DNS is case-insensitive per RFC 4343)
    normalized = lowercase(trim(domain))

    // Step 2: Remove trailing dot if present
    // DNS fully-qualified names may end with a dot (e.g.,
"example.com.")
    if normalized ends with ".":
        normalized = normalized[0 : length(normalized) - 1]

    // Step 3: Punycode encoding for Internationalized Domain Names
(IDN)
    // If the domain contains non-ASCII characters, convert to
    // Punycode representation per RFC 3492 / RFC 5891
    // e.g., "muenchen.de" with u-umlaut becomes "xn--mnchen-3ya.de"
    if containsNonASCII(normalized):
        labels = split(normalized, ".")
        punycodeLabels = []
        for each label in labels:
            if containsNonASCII(label):
                punycodeLabels.append("xn--" + punycodeEncode(label))
            else:
                punycodeLabels.append(label)
        normalized = join(punycodeLabels, ".")

    // Step 4: Validation
    // Domain must contain at least one dot (label + TLD)
    if not contains(normalized, "."):
        raise Error("Invalid domain: must include TLD")

    // Domain must not exceed 253 characters (RFC 1035)
    if length(normalized) > 253:
        raise Error("Invalid domain: exceeds maximum length")

    // Each label must be 1-63 characters, alphanumeric or hyphen,
    // must not start or end with hyphen
    labels = split(normalized, ".")
    for each label in labels:
        if length(label) < 1 or length(label) > 63:
            raise Error("Invalid label length: " + label)
        if not matches(label, "^[a-z0-9]([a-z0-9-]*[a-z0-9])?$"):
            raise Error("Invalid label characters: " + label)
```

```
    // Step 5: Prefix with dollar sign
    return "$" + normalized
```

Uniqueness proof: The function `domainToTokenSymbol` is injective (one-to-one). Given two distinct domain names D1 and D2 where D1 != D2, their normalized forms must also differ (since normalization is a deterministic, reversible process: lowercasing is idempotent on already-lowercase strings, trailing-dot removal is idempotent, and Punycode encoding is a bijection). Since the dollar-sign prefix is applied identically to both, `"$" + normalize(D1) != "$" + normalize(D2)`. Therefore no two distinct domains can produce the same token symbol. The ICANN domain registry guarantees that no two registered domains have the same fully-qualified name, completing the uniqueness proof.

3. On-Chain Domain Token Inscription

Upon successful ownership verification, the system inscribes domain token metadata on the BSV blockchain as a 1-satoshi ordinal inscription. The inscription serves as an immutable, publicly verifiable deed of domain tokenization.

3.1 Inscription Document Structure

The inscription contains a JSON document with the following structure:

```
{
  "p": "$402",
  "op": "publish",
  "type": "domain_verification",
  "title": "<domain_name>",
  "proof": {
    "dns_txt": "<verification_proof>",
    "timestamp": "<ISO_8601_timestamp>",
    "signature": "<optional_cryptographic_signature>"
  },
  "author": "<owner_wallet_address>",
  "parent": "<authority_token_reference>"
}
```

The fields are:

- `p` – Protocol identifier: `$402`, indicating this inscription follows the $402 protocol standard.
- `op` – Operation: `publish`, indicating a new domain token publication.

- `type` – Content type: `domain_verification`, distinguishing this from URL path tokenizations.
- `title` – The domain name being tokenized.
- `proof` – The verification proof artifact, linking the inscription to the ownership verification.
- `author` – The wallet address of the domain's verified owner.
- `parent` – Optional reference to an authority token, establishing the inscription within a hierarchy.

3.2 Inscription Mechanism

The inscription is created as a BSV transaction with the following structure:

1. **Input**: One or more unspent transaction outputs (UTXOs) from the platform's treasury address, providing funding for the inscription and miner fee.
2. **Output 1 (Inscription)**: A 1-satoshi output containing an OP_RETURN script. The script contains: the OP_RETURN opcode (106), the protocol identifier encoded as UTF-8, the content type (`application/json;charset=utf-8`) encoded as UTF-8, and the inscription document encoded as UTF-8.
3. **Output 2 (Change)**: The remaining satoshis minus the miner fee returned to the treasury address.

The transaction is signed using the platform's private key and broadcast to the BSV network. Upon confirmation, the transaction ID serves as the canonical, immutable reference for the domain token.

3.2.1 OP_RETURN Script Byte Layout

The OP_RETURN output script has the following exact byte layout:

```
Byte 0:       0x6a          (OP_RETURN opcode, decimal 106)
Byte 1:       0x04          (OP_PUSHDATA length prefix: 4 bytes follow)
Bytes 2-5:    0x24 0x34 0x30 0x32   (UTF-8 encoding of "$402")
Byte 6:       0x1e          (OP_PUSHDATA length prefix: 30 bytes follow)
Bytes 7-36:   UTF-8 encoding of "application/json;charset=utf-8"
Byte 37:      0x4c          (OP_PUSHDATA1 opcode, next byte is length)
Byte 38:      <length>      (length of JSON document in bytes, 0-255)
                             (if JSON exceeds 255 bytes, use
OP_PUSHDATA2:
                              Byte 37: 0x4d, Bytes 38-39: little-endian
uint16 length)
Bytes 39+:    UTF-8 encoding of the JSON inscription document
```

For JSON documents exceeding 255 bytes (common for domain inscriptions with full proof data), the OP_PUSHDATA2 opcode (0x4d) is used instead, with a two-byte little-endian length prefix. For documents exceeding 65,535 bytes, OP_PUSHDATA4 (0x4e) is used with a four-byte little-endian length prefix.

The total script size for a typical domain token inscription is approximately 300-500 bytes.

3.3 Inscription Verification

Any party can independently verify a domain token inscription by:

1. Retrieving the transaction from the blockchain using the transaction ID.
2. Parsing the OP_RETURN output to extract the JSON inscription document.
3. Verifying that the protocol identifier matches `$402` and the operation is `publish`.
4. Checking that the domain in the inscription matches the claimed domain.
5. Optionally re-verifying the ownership proof by performing the same DNS/HTML/file verification.

This creates a full verification chain: DNS ownership -> verification proof -> on-chain inscription. The chain is auditable by any party without reliance on the DNS-DEX platform.

4. Square-Root Decay Deterministic Pricing

The system prices fractional domain token interests using a deterministic mathematical formula that requires no order book, no price negotiation, and no centralised oracle.

4.1 The Pricing Formula

The price of a single token at any point in time is:

```
price = base_price / sqrt(supply_remaining + 1)
```

Where:

- `base_price` is a constant set at the time of tokenization (denominated in satoshis).
- `supply_remaining` is the number of tokens not yet distributed (total_supply minus tokens_sold).
- `sqrt` is the square root function.
- The `+1` prevents division by zero when supply_remaining reaches zero.

4.2 Pricing Behaviour

The formula produces a monotonically increasing price curve as tokens are distributed:

Tokens Sold	Supply Remaining (of 100,000)	Price (base = 100,000 sats)
0	100,000	317 sats
10,000	90,000	334 sats
50,000	50,000	447 sats
90,000	10,000	1,000 sats
99,000	1,000	3,163 sats
99,900	100	9,950 sats
99,990	10	30,151 sats
99,999	1	70,711 sats
100,000	0	100,000 sats

Key properties:

- **Early buyers are rewarded**: When supply_remaining is large (many tokens remain), the denominator is large and the price is low. The first buyer pays the least. This creates a strong incentive for early participation and rewards those who tokenize or invest in domains before they become popular.
- **Scarcity premium**: As supply_remaining decreases, the price rises. The last tokens are the most expensive, reflecting the scarcity of remaining supply.
- **Deterministic and auditable**: The price at any point is a pure function of two publicly observable values: the base_price (immutably recorded in the on-chain inscription) and the supply_remaining (derived from the total supply minus confirmed purchases). Any party can independently compute the current price without querying the platform.

- **No order book required**: Unlike exchange-based pricing where the price is set by matching buy and sell orders, this formula provides a price at all times regardless of whether there are pending orders. This eliminates the chicken-and-egg problem of bootstrapping a new market with no initial liquidity.
- **Mathematically guaranteed ordering**: Because the square root function is monotonically increasing, and supply_remaining monotonically decreases with each purchase, the price is guaranteed to be lower for earlier buyers and higher for later buyers. This guarantee is mathematical, not contractual, and cannot be circumvented by the platform.

4.3 Batch Pricing

The system supports batch pricing for acquiring multiple tokens in a single transaction. The total cost of acquiring `n` tokens is computed by iterating through each token purchase sequentially:

```
total_cost = sum(base_price / sqrt(supply_remaining - i + 1)) for i = 0
to n-1
```

Each successive token in the batch is priced at the marginal price after the previous token has been notionally purchased (i.e., supply_remaining decreases by 1 for each token in the batch). This ensures that batch purchases pay the same total as sequential individual purchases, preventing arbitrage between batch and individual purchase strategies.

The implementation function `calculateTotalCost(basePriceSats, currentSupply, totalSupply, amount)` returns: the number of tokens, the total cost in satoshis, the average price per token, and the price range (first token price and last token price in the batch).

4.3.1 Batch Pricing Pseudocode with Integer Arithmetic

To avoid floating-point precision errors (which could cause different implementations to compute different prices for the same inputs), the system uses scaled integer arithmetic internally:

Pseudocode:

```
function calculateTotalCost(basePriceSats, currentSupply, totalSupply,
amount):
    // basePriceSats: integer, the base price in satoshis
    // currentSupply: integer, number of tokens already distributed
    // totalSupply: integer, total token supply
    // amount: integer, number of tokens to acquire

    if currentSupply + amount > totalSupply:
        raise Error("Insufficient supply")

    // Use a scaling factor of 10^12 for internal arithmetic
    // to maintain precision without floating-point
    SCALE = 1_000_000_000_000  // 10^12

    totalCostScaled = 0
    supplyRemaining = totalSupply - currentSupply

    for i = 0 to amount - 1:
        // Compute sqrt(supplyRemaining - i + 1) using integer square
root
        // with SCALE precision
        denominator = supplyRemaining - i + 1

        // isqrt computes floor(sqrt(n * SCALE^2)) = floor(sqrt(n) *
SCALE)
        sqrtScaled = isqrt(denominator * SCALE * SCALE)

        // price_i = basePriceSats * SCALE / sqrtScaled
        priceScaled = (basePriceSats * SCALE) / sqrtScaled

        totalCostScaled = totalCostScaled + priceScaled

    // Convert back from scaled representation
    // Round to nearest satoshi (integer)
    totalCostSats = (totalCostScaled + SCALE / 2) / SCALE

    firstPrice = basePriceSats * SCALE / isqrt((supplyRemaining + 1) *
SCALE * SCALE)
    firstPriceSats = (firstPrice + SCALE / 2) / SCALE

    lastPrice = basePriceSats * SCALE / isqrt((supplyRemaining - amount
+ 2) * SCALE * SCALE)
    lastPriceSats = (lastPrice + SCALE / 2) / SCALE

    averagePriceSats = totalCostSats / amount
```

```
    return {
        tokens: amount,
        totalCostSats: totalCostSats,
        averagePriceSats: averagePriceSats,
        priceRange: { first: firstPriceSats, last: lastPriceSats }
    }

// Integer square root using Newton's method
function isqrt(n):
    if n < 0: raise Error("Negative input")
    if n == 0: return 0
    x = n
    y = (x + 1) / 2
    while y < x:
        x = y
        y = (x + n / x) / 2
    return x
```

The scaling factor of 10^12 ensures that individual token prices computed by any conforming implementation agree to within 1 satoshi, regardless of the programming language or platform used.

4.4 Inverse Calculation

The system also supports inverse calculation: given a spending budget in satoshis, compute the maximum number of tokens acquirable within that budget. This is implemented by iteratively purchasing tokens at the current marginal price until the budget is exhausted. The function returns: the number of tokens acquired, the total cost, and the remaining unspent satoshis.

Pseudocode:

```
function calculateTokensForBudget(basePriceSats, currentSupply,
totalSupply, budgetSats):
    SCALE = 1_000_000_000_000
    tokensAcquired = 0
    totalSpentScaled = 0
    supplyRemaining = totalSupply - currentSupply

    while tokensAcquired < supplyRemaining:
        denominator = supplyRemaining - tokensAcquired + 1
        sqrtScaled = isqrt(denominator * SCALE * SCALE)
        priceScaled = (basePriceSats * SCALE) / sqrtScaled

        // Check if we can afford one more token
        projectedTotal = totalSpentScaled + priceScaled
        projectedTotalSats = (projectedTotal + SCALE / 2) / SCALE

        if projectedTotalSats > budgetSats:
            break  // budget exhausted

        totalSpentScaled = projectedTotal
        tokensAcquired = tokensAcquired + 1

    totalSpentSats = (totalSpentScaled + SCALE / 2) / SCALE
    remainingSats = budgetSats - totalSpentSats

    return {
        tokens: tokensAcquired,
        totalCostSats: totalSpentSats,
        remainingSats: remainingSats
    }
```

5. Micropayment Revenue Distribution

The system distributes revenue from a domain's web traffic to holders of tokens representing that domain.

5.1 Revenue Generation

Domain visitors interact with content gated by the HTTP 402 payment protocol. When a visitor accesses gated content on a tokenized domain, the server responds with HTTP 402 Payment Required and the visitor's client (which may be a browser extension, a wallet application, or an autonomous software agent) submits a micropayment to access the content. Typical micropayment amounts are 1-5 cents per access event.

5.2 Revenue Split

Revenue from each micropayment is split according to a declared ratio:

Recipient	Default Share	Purpose
Domain Owner	80%	Compensation for domain ownership and content creation
Distribution Pool	20%	Revenue available for distribution to token holders

The split ratio is declared at the time of tokenization and recorded in the on-chain inscription. It may be modified by the domain owner through a subsequent on-chain transaction, but any modification is publicly visible and auditable.

5.3 Token Holder Dividends

The distribution pool is allocated to token holders proportionally to their token holdings:

- A holder with 10% of the total token supply receives 10% of the distribution pool.
- A holder with 0.1% of the total token supply receives 0.1% of the distribution pool.
- A holder with zero tokens receives nothing.

Distribution may occur per-transaction (each micropayment triggers an immediate micro-distribution), per-epoch (accumulated revenue is distributed at regular intervals), or on-demand (holders claim accumulated dividends through a claim transaction).

5.3.1 Revenue Split and Dividend Computation Pseudocode

The following pseudocode describes the complete revenue processing pipeline for a single micropayment event:

```
function processRevenueEvent(paymentTxId, domainToken,
paymentAmountSats):
    // Step 1: Look up the domain token's on-chain inscription
    inscription = lookupInscription(domainToken.inscriptionTxId)
    ownerAddress = inscription.author
    splitRatio = inscription.ownerSharePercent  // e.g., 80

    // Step 2: Compute the split
    ownerShareSats = floor(paymentAmountSats * splitRatio / 100)
    poolShareSats = paymentAmountSats - ownerShareSats  // avoids
rounding loss

    // Step 3: Send owner's share immediately
    sendPayment(ownerAddress, ownerShareSats)

    // Step 4: Record pool accumulation
    // The pool accumulates until a distribution event
    domainToken.accumulatedPoolSats += poolShareSats

    // Step 5: Record the revenue event on-chain
    revenueRecord = {
        "p": "$402",
        "op": "revenue",
        "type": "domain_revenue",
        "domain": domainToken.title,
        "paymentTx": paymentTxId,
        "totalSats": paymentAmountSats,
        "ownerSats": ownerShareSats,
        "poolSats": poolShareSats,
        "accumulatedPoolSats": domainToken.accumulatedPoolSats,
        "timestamp": currentISO8601()
    }
    inscribeOnChain(revenueRecord)

    return { ownerShareSats, poolShareSats }


function distributePoolDividends(domainToken):
    // Called per-epoch or on-demand
    poolSats = domainToken.accumulatedPoolSats
    if poolSats == 0:
        return []

    // Step 1: Get all token holders and their balances
    holders = getTokenHolders(domainToken.inscriptionTxId)
    // holders is a list of { address, balance } objects
```

```
    // totalSupply is the sum of all holder balances

    totalSupply = domainToken.totalSupply
    distributions = []

    // Step 2: Compute each holder's share using integer arithmetic
    // to avoid floating-point and ensure total distributed == poolSats
    distributedSoFar = 0
    sortedHolders = sortByBalance(holders, descending=true)

    for i = 0 to length(sortedHolders) - 1:
        holder = sortedHolders[i]

        if i == length(sortedHolders) - 1:
            // Last holder gets the remainder (avoids rounding dust)
            holderShareSats = poolSats - distributedSoFar
        else:
            // Integer division: floor(poolSats * balance / totalSupply)
            holderShareSats = floor(poolSats * holder.balance /
totalSupply)

        if holderShareSats > 0:
            distributions.append({
                address: holder.address,
                amountSats: holderShareSats,
                balance: holder.balance,
                percentOfSupply: holder.balance * 100 / totalSupply
            })
            distributedSoFar += holderShareSats

    // Step 3: Execute payments and record on-chain
    for each dist in distributions:
        sendPayment(dist.address, dist.amountSats)

    // Step 4: Record the distribution event on-chain
    distributionRecord = {
        "p": "$402",
        "op": "dividend",
        "type": "domain_dividend",
        "domain": domainToken.title,
        "epochPoolSats": poolSats,
        "recipientCount": length(distributions),
        "distributions": distributions,
        "timestamp": currentISO8601()
    }
    inscribeOnChain(distributionRecord)
```

```
    // Step 5: Reset the accumulated pool
    domainToken.accumulatedPoolSats = 0

    return distributions
```

Integer arithmetic guarantee: The algorithm uses `floor(poolSats * balance / totalSupply)` for all holders except the last, who receives the remainder. This ensures that the total distributed exactly equals the pool amount with no satoshis lost to rounding errors.

5.4 On-Chain Revenue Accounting

Revenue events are indexed on-chain, providing transparent accounting:

- Each micropayment is recorded as a blockchain transaction.
- Revenue data is aggregated per domain, per time period, and per token holder.
- Any party can independently audit the revenue stream and dividend distribution by examining the blockchain.
- The x402 payment indexing mechanism enables real-time revenue tracking without relying on the platform's own reporting.

The revenue distribution mechanism transforms domain tokens from speculative assets into income-producing investments. Token holders receive ongoing economic returns from the domain's web traffic, creating a sustained economic incentive for holding and promoting the domain.

6. Sovereign Bridge Progressive Decentralisation Architecture

The system provides a three-phase architecture for progressively decentralising domain content delivery, enabling domain owners to balance performance and censorship resistance.

6.1 Phase 1: Cloud-Primary

In the initial phase, domain content is served entirely from cloud infrastructure (e.g., Vercel, AWS, Cloudflare). This provides:

- Standard web performance (low latency, high throughput, global CDN distribution).
- Familiar development and deployment workflows.
- No requirement for blockchain storage or retrieval.

During Phase 1, the domain's token metadata is inscribed on-chain (as described in Section 3), establishing ownership proof and tokenization parameters. The content itself, however, is

served from cloud infrastructure. The on-chain inscription serves as a “title deed” – it proves the tokenization occurred and records the parameters, but it does not store the content.

6.1.1 Phase 1 DNS Configuration

In Phase 1, the domain’s DNS is configured with a standard CNAME or A record pointing to the cloud provider:

```
;; Example DNS configuration for Phase 1
example.com.    IN  CNAME   cname.vercel-dns.com.
;; Or for AWS CloudFront:
example.com.    IN  CNAME   d1234567890.cloudfront.net.
;; Or for Cloudflare:
example.com.    IN  A       104.21.xxx.xxx
```

The DNS-DEX verification TXT record coexists alongside these records:

```
_dnsdex.example.com.  IN  TXT  "dnsdex-
verify=a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6"
```

No modification to the domain’s content delivery infrastructure is required for Phase 1 tokenization. The domain operates identically to a non-tokenized domain from the visitor’s perspective.

6.2 Phase 2: Hybrid Cloud-Blockchain

In the second phase, the system periodically snapshots domain content and archives it to blockchain-based distributed storage (ordFS – ordinal file system on BSV, or equivalent content-addressable blockchain storage).

- Snapshots are taken at configurable intervals (e.g., daily, weekly, on content change).
- Each snapshot creates an on-chain record containing: the snapshot timestamp, the content hash, and the blockchain storage reference.
- The on-chain snapshots serve as an “insurance policy” – verifiable, immutable backups of the domain’s content at specific points in time.
- During Phase 2, content is still served from cloud infrastructure for performance. The on-chain snapshots are used only for verification and backup.

This phase creates a dual-track content delivery system: cloud for speed, blockchain for permanence. If the cloud provider censors the content or experiences an outage, the on-chain

snapshots provide a verifiable record of what the content was, and a retrievable backup of the content itself.

6.2.1 Content Snapshot Process

The snapshot process operates as follows:

Pseudocode:

```
function createContentSnapshot(domain, domainTokenTxId):
    // Step 1: Crawl the domain's public content
    // Starting from the homepage, follow internal links up to a
    // configurable depth (default: 3 levels) and size limit (default:
50 MiB)
    pages = crawlDomain("https://" + domain, maxDepth=3,
maxTotalSize=52428800)

    // Step 2: Create a content bundle
    // Each page is stored as: { path, contentType, body, headers }
    bundle = {
        domain: domain,
        crawlTimestamp: currentISO8601(),
        pages: pages,
        totalSize: sum(page.body.length for page in pages)
    }

    // Step 3: Compute content hash of the entire bundle
    bundleBytes = serialize(bundle)  // deterministic JSON serialization
    contentHash = SHA256(bundleBytes)

    // Step 4: Inscribe the bundle on-chain using ordFS
    // For large bundles, split into chunks of 100 KiB each
    CHUNK_SIZE = 102400  // 100 KiB
    chunkTxIds = []
    for i = 0 to ceil(length(bundleBytes) / CHUNK_SIZE) - 1:
        chunk = bundleBytes[i * CHUNK_SIZE : (i + 1) * CHUNK_SIZE]
        txId = inscribeChunk(chunk, index=i,
totalChunks=ceil(length(bundleBytes) / CHUNK_SIZE))
        chunkTxIds.append(txId)

    // Step 5: Create a snapshot manifest inscription
    manifest = {
        "p": "$402",
        "op": "snapshot",
        "type": "domain_snapshot",
        "domain": domain,
        "domainTokenTx": domainTokenTxId,
        "contentHash": hex(contentHash),
        "totalSize": length(bundleBytes),
        "chunkCount": length(chunkTxIds),
        "chunks": chunkTxIds,
        "timestamp": currentISO8601()
    }
    manifestTxId = inscribeOnChain(manifest)
```

```
    return {
        manifestTxId: manifestTxId,
        contentHash: hex(contentHash),
        chunkCount: length(chunkTxIds),
        totalSize: length(bundleBytes)
    }
```

6.3 Phase 3: On-Chain Fallback

In the third phase, the system provides automatic fallback to blockchain-based content delivery when cloud infrastructure becomes unavailable:

- Protocol resolvers (software that knows how to resolve domain tokens to content) monitor the cloud endpoint for availability.
- If the cloud endpoint returns errors or becomes unreachable, the resolver automatically redirects content requests to the most recent on-chain snapshot.
- On-chain content delivery is slower than cloud delivery but is censorship-resistant – the content is distributed across the blockchain network and cannot be removed by any single party.
- Token holders may vote (via a governance mechanism weighted by token holdings) to redirect the domain to alternative infrastructure, such as a new cloud provider, an IPFS gateway, or a different blockchain storage system.

6.3.1 Protocol Resolver Fallback Algorithm

The protocol resolver implements the following fallback logic:

Pseudocode:

```
function resolveContent(domain, path):
    // Step 1: Look up the domain token inscription
    domainToken = lookupDomainToken(domain)
    if domainToken is null:
        return { status: 404, body: "Domain not tokenized" }

    // Step 2: Determine the domain's current phase
    latestSnapshot = getLatestSnapshot(domainToken.inscriptionTxId)

    // Step 3: Attempt cloud delivery first
    cloudUrl = "https://" + domain + path
    try:
        response = HTTP_GET(cloudUrl, timeout=5000)
        if response.status >= 200 and response.status < 500:
            // Cloud is healthy (even 4xx errors mean the server is up)
            return response
    catch (networkError):
        // Cloud is unreachable -- fall through to on-chain fallback
        pass

    // Step 4: If cloud fails, try the most recent on-chain snapshot
    if latestSnapshot is not null:
        snapshotContent = retrieveSnapshot(latestSnapshot.manifestTxId)
        if snapshotContent is not null:
            // Find the requested path within the snapshot
            page = snapshotContent.pages.find(p => p.path == path)
            if page is not null:
                return {
                    status: 200,
                    headers: {
                        "Content-Type": page.contentType,
                        "X-DNSDex-Source": "on-chain",
                        "X-DNSDex-Snapshot":
latestSnapshot.manifestTxId,
                        "X-DNSDex-Snapshot-Date":
latestSnapshot.timestamp
                    },
                    body: page.body
                }

    // Step 5: Check for governance redirect
    redirect = getGovernanceRedirect(domainToken.inscriptionTxId)
    if redirect is not null:
        return HTTP_REDIRECT(302, redirect.targetUrl + path)
```

```
    return { status: 503, body: "Content temporarily unavailable" }
```

The resolver adds `X-DNSDex-Source` headers to indicate whether content was served from cloud or from on-chain storage, providing transparency to clients about the content delivery path.

6.4 Progressive Migration Path

The three phases are not mutually exclusive – they represent a progression. A domain may remain in Phase 1 indefinitely if the owner is satisfied with cloud-only delivery. A domain may operate in Phase 2 indefinitely, maintaining cloud delivery with blockchain backups. Phase 3 activates automatically when cloud delivery fails, and may be reversed when cloud delivery is restored.

This architecture solves the false dichotomy between centralised performance and decentralised censorship resistance. Domain owners need not choose one or the other – they begin with centralised performance and progressively add decentralised resilience at their own pace.

7. Integration with Existing Protocols

The DNS-DEX system operates on top of and extends existing protocol infrastructure:

- **$401 (Identity)**: Provides verified identity for domain tokenizers. The `author` field in the on-chain inscription references the tokenizer's $401 identity chain, creating an auditable link between a verified identity and the tokenization action. Identity verification levels (Level 1 through Level 4+) provide graduated assurance of the tokenizer's real-world identity.
- **$402 (Commerce)**: Provides the underlying token market mechanics, HTTP 402 payment protocol, algorithmic pricing, and the monetary signalling convention (dollar-prefix). The DNS-DEX system extends $402 from URL path tokenization to domain-level tokenization. The on-chain inscription uses the `$402` protocol identifier, ensuring compatibility with the broader $402 ecosystem.
- **$403 (Securities)**: For domain tokens that may constitute securities (e.g., tokens offering revenue sharing that may qualify as investment contracts under securities law), the $403 protocol provides KYC-gated access controls. Domain tokens may be configured to require $401 Level 2+ identity verification and $403 compliance gating before acquisition.

- **Proof of Indexing**: The verification and inscription work performed during domain tokenization constitutes verifiable indexing work. Nodes that perform domain verification and store domain token data may commit this work to the Proof of Indexing mining mechanism, earning $402 utility tokens for their contribution to the domain token index.

8. Implementation Architecture

A reference implementation comprises:

Domain Verification Engine: A serverless-compatible module that performs multi-proof verification using DNS-over-HTTPS, HTML meta tag parsing, and well-known file retrieval. The module accepts a domain name and a verification code, attempts all three methods in fallback order, and returns a structured verification result.

Inscription Engine: A module that constructs and broadcasts BSV transactions containing domain token inscriptions. The module manages UTXO selection, transaction construction, OP_RETURN script generation, transaction signing, and broadcast via the WhatsOnChain API or equivalent blockchain API.

Pricing Engine: A module that computes square-root decay prices for domain token purchases. The module supports single-token pricing, batch pricing, inverse calculation (tokens for budget), and price schedule generation for visualisation.

Revenue Distribution Engine: A module that indexes micropayment revenue from x402 transactions on tokenized domains, computes proportional dividends for token holders, and executes or records dividend distributions.

Sovereign Bridge: A module that manages the progressive decentralisation lifecycle: monitoring cloud endpoint availability, creating periodic content snapshots for on-chain storage, and activating fallback content delivery from blockchain storage when cloud delivery fails.

Marketplace Interface: A web-based interface enabling domain owners to initiate tokenization, domain investors to browse and acquire domain tokens, and all parties to view pricing curves, revenue data, and verification status. The interface presents verification instructions for all three methods (DNS, HTML, file) with step-by-step guidance tailored to popular DNS providers.

Brief Description of Drawings

The following drawings would accompany this application:

- **Figure 1** – System architecture diagram showing the complete DNS-DEX system: multi-proof verification engine, inscription engine, pricing engine, revenue distribution engine, sovereign bridge, and marketplace interface, with data flows between components.
- **Figure 2** – Multi-proof verification flow showing the three verification methods (DNS TXT via DNS-over-HTTPS, HTML meta tag via HTTP GET, well-known file via HTTP GET) executed in fallback order, with decision points and result aggregation.
- **Figure 3** – Domain-as-token-symbol mapping showing DNS domain names being transformed into token symbols by the deterministic `$<domain.tld>` convention, with examples across multiple TLDs.
- **Figure 4** – On-chain inscription structure showing the BSV transaction structure with OP_RETURN output containing the $402 protocol identifier, content type, and JSON inscription document, with field-level annotation of the inscription metadata.
- **Figure 5** – Square-root decay pricing curve showing token price on the y-axis and supply remaining on the x-axis, with annotated regions for early-buyer advantage, mid-market zone, and scarcity premium zone, and the mathematical formula `price = base_price / sqrt(supply_remaining + 1)`.
- **Figure 6** – Revenue distribution flow showing: visitor micropayment via HTTP 402 -> payment split between domain owner (80%) and distribution pool (20%) -> proportional distribution from pool to token holders based on their holdings, with on-chain accounting.
- **Figure 7** – Sovereign bridge progressive decentralisation showing the three phases: Phase 1 (cloud-primary with on-chain title deed), Phase 2 (hybrid with periodic on-chain snapshots), and Phase 3 (automatic fallback to on-chain content delivery when cloud is unavailable), with monitoring and governance components.
- **Figure 8** – Complete tokenization workflow showing the end-to-end process: domain input -> verification code generation -> multi-proof verification (DNS/HTML/file) -> on-chain inscription -> marketplace listing -> pricing engine -> token acquisition -> revenue distribution -> sovereign bridge lifecycle.

Initial Claims

Note: These claims are provided in sketch form for the purposes of establishing a priority date. Formal claims will be drafted and filed within 12 months in accordance with UKIPO rules.

Claim 1 – Multi-Proof Domain Ownership Verification for Tokenization

A method of verifying domain name ownership for the purpose of creating a blockchain token representing the domain, the method comprising:

a. generating a unique cryptographic verification code that is specific to the domain name and the requesting party's wallet address;

b. verifying domain ownership through one or more of: a DNS TXT record query performed via DNS-over-HTTPS by issuing standard HTTPS requests to public DNS resolvers to enable verification in serverless computing environments where system-level DNS utilities are unavailable, an HTML meta tag embedded in the domain's web content and retrieved via standard HTTP request, or a standardised well-known file at a predetermined path retrieved via standard HTTP request;

c. attempting the verification methods in a fallback order, returning a successful result upon the first method that confirms ownership;

d. upon successful verification, inscribing domain token metadata on a blockchain as an immutable ordinal inscription containing the verification proof, domain name, token parameters, and timestamp;

wherein the multi-proof verification enables domain ownership confirmation across diverse computing environments including serverless platforms that lack system-level DNS resolution capabilities, and wherein the verification creates a cryptographic chain from DNS ownership through verification proof to on-chain token inscription.

Claim 2 – Domain-as-Token-Symbol Convention

A system for creating blockchain tokens representing domain names, wherein:

a. the token symbol is derived directly from the domain name by prefixing the full domain name (including top-level domain) with a monetary signalling character, producing a symbol of the form `$<domain.tld>`;

b. symbol uniqueness is guaranteed by the uniqueness of domain names in the Domain Name System, such that no external symbol registry is required to prevent collisions;

c. the token symbol is human-readable and self-documenting, identifying the underlying asset without reference to an external registry or lookup table;

d. the convention extends a URL path tokenization convention upward to the domain level, creating a hierarchical namespace where `$domain.tld` identifies the domain token and `$domain.tld/$path` identifies content tokens within that domain;

wherein each tokenized domain produces exactly one token symbol of the form `$<domain.tld>` that is globally unique and directly maps to the domain name.

Claim 3 – Square-Root Decay Pricing for Domain Tokens

A method of pricing fractional interests in a tokenized domain name, the method comprising:

a. establishing a base price denominated in a specified currency unit and a total supply at the time of tokenization, with both values immutably recorded in an on-chain inscription;

b. computing the current price of a single token as a function of the remaining supply using a square-root decay formula `price = base_price / sqrt(supply_remaining + 1)`, where supply_remaining is the total supply minus the number of tokens already distributed;

c. enabling any party to independently compute the current price from publicly observable state without querying the platform or any centralised oracle;

d. computing the total cost of acquiring multiple tokens in a batch by sequentially applying the pricing formula with decremented supply for each successive token;

wherein the pricing is deterministic, requires no order book or price negotiation, and mathematically guarantees that early acquirers receive a lower price than later acquirers through the monotonically increasing relationship between tokens distributed and price.

Claim 4 – Micropayment Revenue Distribution to Domain Token Holders

A method of distributing revenue from a domain's web traffic to holders of tokens representing that domain, the method comprising:

a. receiving micropayments from visitors accessing gated content on the tokenized domain via an HTTP 402 payment protocol;

b. splitting the received payments according to a declared ratio between the domain owner and a distribution pool, the ratio being recorded in the domain's on-chain token inscription;

c. distributing the distribution pool to token holders proportionally to their token holdings, such that a holder with x% of the total supply receives x% of the distribution pool;

d. recording revenue events on a blockchain for transparent, independently auditable accounting;

wherein token holders receive ongoing economic returns from the domain's web traffic proportional to their ownership stake, and wherein all revenue flows are publicly verifiable

on-chain without reliance on the platform's own reporting.

Claim 5 – Progressive Decentralisation of Tokenized Domain Content

A system for serving content associated with a tokenized domain name through progressive decentralisation, the system comprising:

a. a primary content delivery layer using cloud infrastructure for standard web performance, with domain token metadata inscribed on-chain as a title deed;

b. a periodic archival mechanism that creates snapshots of domain content and stores them on blockchain-based distributed storage, each snapshot including a timestamp, content hash, and storage reference;

c. an automatic fallback mechanism comprising a monitoring component that detects cloud infrastructure unavailability and a resolver component that redirects content requests to the most recent on-chain content snapshot;

d. a governance mechanism enabling token holders to vote, weighted by token holdings, on infrastructure changes including redirecting the domain to alternative delivery infrastructure;

wherein content delivery progressively migrates from centralised to decentralised infrastructure through three phases (cloud-primary, hybrid cloud-blockchain, on-chain fallback) while maintaining content availability at each phase, and wherein domain owners may progress through phases at their own pace without disrupting existing content delivery.

Claim 6 – Domain Name Fractionalisation via Blockchain Tokens

A system for fractionalising economic interests in a domain name, the system comprising:

a. a multi-proof verification mechanism confirming the tokenizer's ownership of the domain name through one or more of DNS TXT record verification via DNS-over-HTTPS, HTML meta tag verification, and well-known file verification, with fallback ordering;

b. an inscription engine creating an immutable blockchain ordinal record binding the domain name to a token with a specified total supply, with a token symbol derived directly from the domain name by the convention `$<domain.tld>`;

c. a pricing engine computing the current price of each token unit as a deterministic function of remaining supply using a square-root decay formula, enabling any party to independently compute the price from publicly observable state;

d. a revenue distribution mechanism splitting micropayment income from the domain's web traffic between the domain owner and token holders, with proportional distribution to holders and on-chain revenue accounting;

e. a marketplace enabling token holders to trade domain tokens on secondary markets, with transfer inscriptions recording each trade on-chain;

f. a sovereign bridge architecture enabling progressive decentralisation from cloud-primary content delivery through hybrid cloud-blockchain to full on-chain fallback;

wherein the domain name becomes a fractionalised economic asset with transparent deterministic pricing, verifiable ownership, automated revenue distribution, and censorship-resistant content delivery, without transferring DNS control from the domain owner to any other party.

Abstract

A system and method for tokenizing domain names on a blockchain, enabling fractional ownership and automated revenue distribution. Domain ownership is verified through a multi-proof bundle comprising DNS TXT records queried via DNS-over-HTTPS (enabling verification in serverless computing environments where system-level DNS utilities are unavailable), HTML meta tags, and well-known files, attempted in fallback order with any single proof sufficing. Verified domains are inscribed as blockchain ordinals with the token symbol derived directly from the domain name using the convention `$<domain.tld>`, ensuring global uniqueness guaranteed by DNS uniqueness without requiring an external symbol registry. Token pricing follows a square-root decay formula (`price = base_price / sqrt(supply_remaining + 1)`) that deterministically rewards early acquirers with lower prices and creates a scarcity premium for later acquirers, with pricing independently computable from publicly observable state. Micropayment revenue from domain web traffic, collected via the HTTP 402 payment protocol, is distributed to token holders proportionally to their holdings through an on-chain accounting mechanism. A sovereign bridge architecture enables progressive decentralisation from cloud-primary delivery through hybrid cloud-blockchain (with periodic on-chain content snapshots) to full on-chain fallback when cloud infrastructure becomes unavailable, with token-holder governance for infrastructure decisions. The system fractionalises domain ownership without transferring DNS control, creating a marketplace for domain-backed tokens with deterministic pricing, automated dividends, and censorship-resistant content delivery.

Document prepared for UKIPO filing. Priority date to be established upon submission. Applicant: The Bitcoin Corporation Ltd Inventor: Richard Boase Date of preparation: 2 March 2026