Bitcoin OS: A Distributed Operating System on Bitcoin

Technical Whitepaper v1.0

The Bitcoin Corporation LTD (UK Company No. 16735102)

February 2026

Abstract

Bitcoin OS is a distributed operating system where the kernel, file system, identity layer, payment infrastructure, and application suite all operate on the Bitcoin SV blockchain. Unlike conventional operating systems that depend on centralised servers and proprietary cloud infrastructure, Bitcoin OS uses on-chain inscriptions as its file system (ORDFS), cryptographic identity chains as its authentication layer (Path 401), micropayment channels as its resource allocation mechanism (Path 402), and smart contracts as its access control system (Path 403).

This paper describes the architecture required to unify 50+ open source Bitcoin applications into a single coherent operating system, where every app shares identity, payments, storage, and inter-process communication through a common system bus backed by the Bitcoin blockchain.

Table of Contents

---

1. Introduction

The Bitcoin protocol, as originally designed, is a complete system for electronic transactions that extends far beyond simple value transfer. Bitcoin Script is a Turing-complete programming language capable of expressing arbitrary computation. One-satoshi outputs can carry inscribed data — text, images, HTML, JavaScript, entire applications — permanently recorded in the blockchain's immutable ledger.

Bitcoin OS takes this capability to its logical conclusion: if applications can be inscribed on Bitcoin and served via HTTP, if identity can be anchored to on-chain cryptographic proofs, if payments are native to the protocol, and if access control

can be enforced by smart contracts — then the blockchain itself can serve as the foundation for a complete operating system.

The Bitcoin Apps Suite (github.com/bitcoin-apps-suite) currently comprises 50 open source applications spanning productivity, media, finance, social networking, developer tools, and infrastructure. Each application is independently deployed and independently functional. This paper describes the architecture that transforms this collection into a unified operating system.

---

2. Problem Statement

2.1 Current State

The existing Bitcoin OS implementation consists of:

- A **desktop shell** built with Next.js that renders a taskbar, dock, sidebar, and window management system
- **50 applications**, each a standalone Next.js deployment on Vercel, loaded into the shell via `<iframe>` elements pointing at separate domains (e.g., `bitcoin-writer.vercel.app`, `bitcoin-email.vercel.app`)
- **Three postMessage types** constituting the entire inter-process communication: `app-ready`, `os-config`, and `navigate-home`
- **Six competing wallet implementations** (HandCash, Yours Wallet, MetaNet Desktop, MockBRC100, AdaptiveMetanetClient, WalletClient) — none unified
- An **authentication system** that functions within the shell but is invisible to iframed applications due to cross-origin isolation

2.2 Fundamental Gaps

No shared identity. When a user authenticates in Bitcoin Writer, Bitcoin Email has no knowledge of that session. Each app must independently manage user state.

No shared storage. Files created in Bitcoin Drive are inaccessible to Bitcoin Writer. There is no common file system abstraction.

No shared payments. Each app that requires a Bitcoin transaction must independently negotiate wallet access, construct transactions, and manage UTXOs.

No inter-app communication. Bitcoin Email cannot compose a message referencing a Bitcoin Drive file. Bitcoin Calendar cannot create an event from a Bitcoin Jobs listing. The apps are islands.

No on-chain presence. The applications run on Vercel's infrastructure. If Vercel becomes unavailable, every application becomes unavailable. The blockchain — the most resilient infrastructure ever built — hosts none of the application logic.

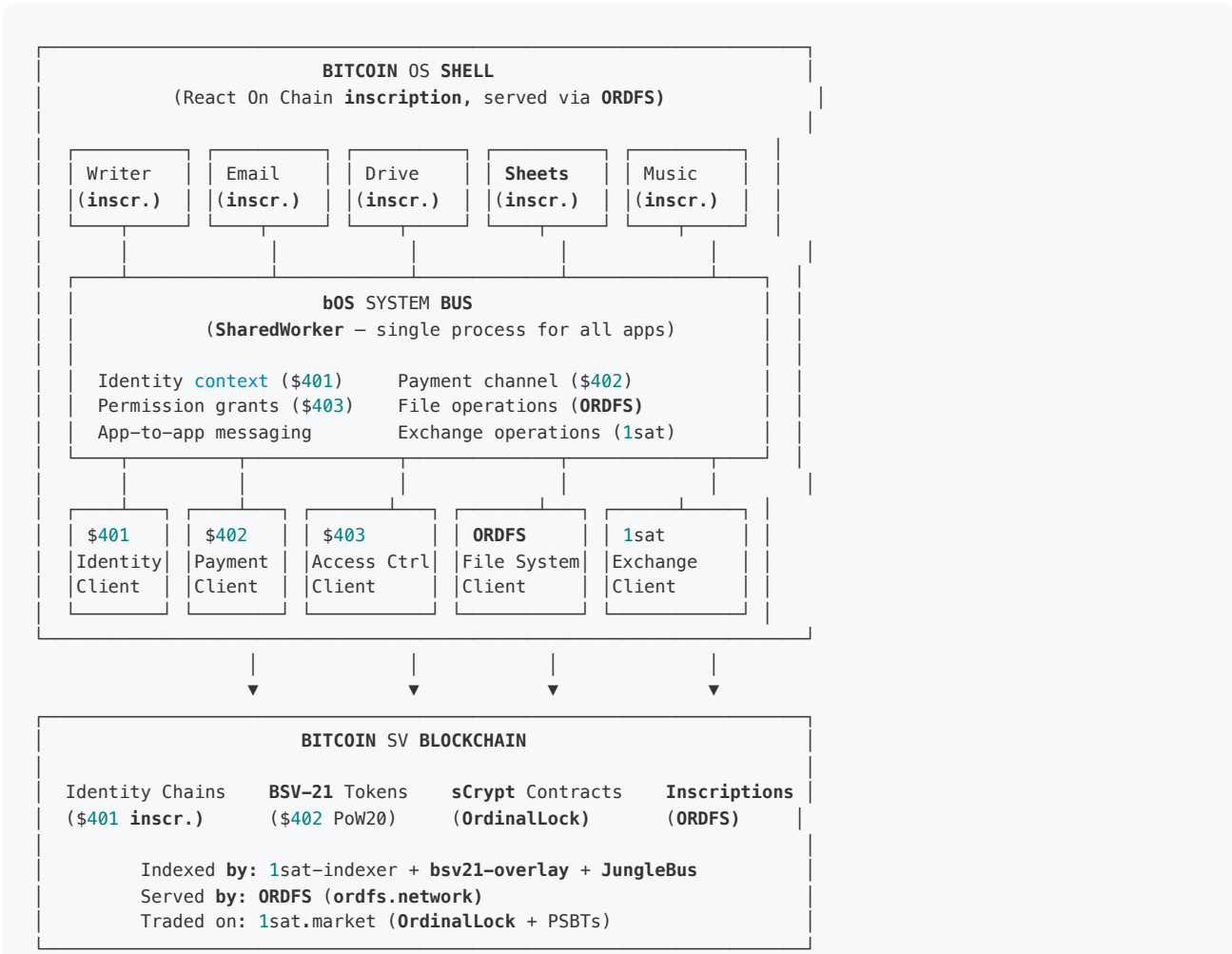
2.3 Design Objective

Build a system where:

1. Every application is inscribed on Bitcoin and served from the blockchain
2. All applications share a single identity, wallet, file system, and communication bus
3. The protocol stack (401/402/403) provides the system services that a traditional OS kernel would provide
4. The entire system can be reconstructed from nothing but a Bitcoin node and an ORDFS server

---

3. Architecture Overview



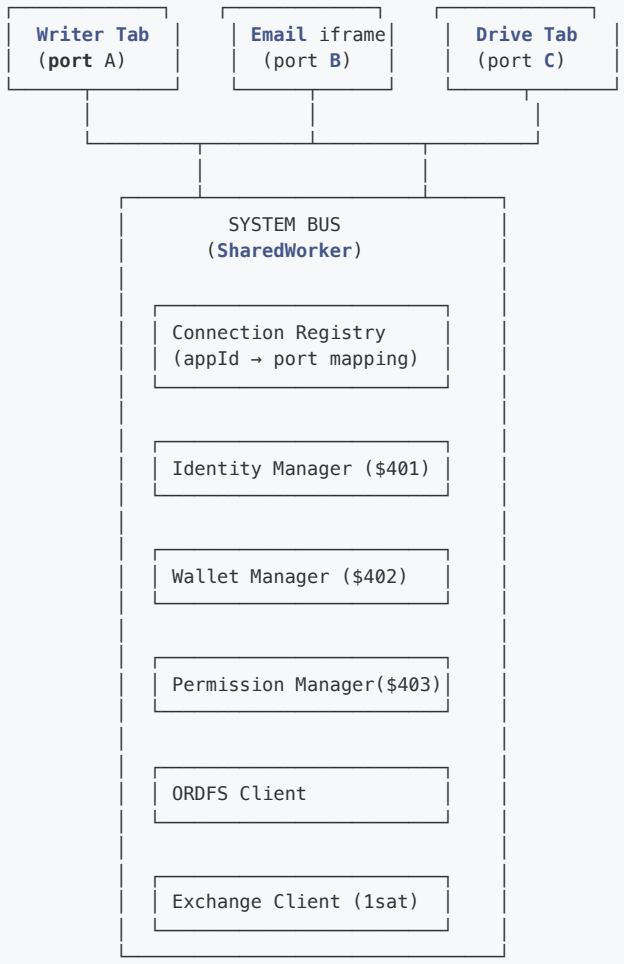
The architecture maps traditional operating system concepts to Bitcoin primitives:

OS Concept	Traditional OS	Bitcoin OS
Kernel	Linux/Windows NT	System Bus (SharedWorker)
File System	ext4 / NTFS / APFS	ORDFS (on-chain inscriptions)
User Identity	/etc/passwd, LDAP	Path 401 (identity chains)
Resource Allocation	CPU scheduler	Path 402 (micropayments)
Access Control	chmod, ACLs	Path 403 (sCrypt contracts)
Package Manager	apt, npm, brew	BitGit (`bit push`)
IPC	Unix sockets, D-Bus	System Bus message passing
App Store	App Store, Play Store	1sat.market (OrdinalLock)
Process Isolation	Containers, VMs	Per-app sCrypt permissions

4. Layer 1: The System Bus

4.1 SharedWorker Architecture

The System Bus is a SharedWorker — a single JavaScript process shared across all browser contexts (tabs, iframes, windows) connected to the same origin. When Bitcoin OS boots, it spawns the SharedWorker. Every app that loads connects to it via a MessagePort.



4.2 System API

Every application imports a lightweight client library (`@bitcoin-os/kernel`) that connects to the SharedWorker and exposes a typed API:

Identity (backed by $401)

```
bOS.identity.whoami()           // Returns current $401 identity
bOS.identity.sign(data)         // Sign with identity key
bOS.identity.verify(sig, pubkey) // Verify a signature
bOS.identity.strength()         // Identity strength (1-4+)
```

Payments (backed by $402)

```
bOS.pay.send(address, satoshis)     // Send BSV
bOS.pay.balance()                    // Wallet balance
bOS.pay.mintToken(appId, metadata)   // Mint BRC-100 token
bOS.pay.transferToken(tokenId, to)   // Transfer token
bOS.pay.history()                    // Transaction history
```

File System (backed by ORDFS)

```
bOS.fs.read(inscriptionId)          // Read inscription content
bOS.fs.write(content, contentType)  // Inscribe content on-chain
bOS.fs.list(query)                  // Query indexed inscriptions
bOS.fs.resolve(domain)              // Resolve ORDFS domain
```

Access Control (backed by $403)

```
bOS.access.check(resource)          // Check permission
bOS.access.request(appId, scope)    // Request access grant
bOS.access.grant(appId, scope)      // Grant access
bOS.access.revoke(appId, scope)     // Revoke access
```

Exchange (backed by 1sat.market)

```
bOS.exchange.list(ordinalId, price)  // List ordinal for sale
bOS.exchange.buy(listingUtxo)        // Purchase listing
bOS.exchange.cancel(listingUtxo)     // Cancel listing
bOS.exchange.bid(ordinalId, price)   // Place bid
bOS.exchange.listings(query)         // Browse listings
```

Inter-Process Communication

```
bOS.send(targetAppId, message)       // Send to specific app
bOS.broadcast(message)               // Broadcast to all apps
bOS.on('message', callback)          // Listen for messages
bOS.on('identity-changed', callback) // System events
bOS.on('payment-received', callback)
```

4.3 Why SharedWorker

A SharedWorker solves the fundamental problem of the current architecture: cross-origin iframe isolation. When Bitcoin Email runs in an iframe on `bitcoin-email.vercel.app`, it cannot access `localStorage`, cookies, or JavaScript objects from the parent `bitcoin-os.website` frame.

The SharedWorker runs in a separate process with its own global scope. Any page from any origin can connect to it, provided they know the worker's URL. The System Bus worker is loaded from a well-known inscription on Bitcoin (served via ORDFS), making it origin-independent and immutable.

---

5. Layer 2: On-Chain Applications

5.1 ORDFS (Ordinal File System)

ORDFS is an HTTP server that bridges Bitcoin inscriptions with web browsers. It serves the contents of any inscription as a standard HTTP response, resolving content by inscription ID (transaction hash + output index).

DNS Resolution Flow:

1. A domain's A/CNAME record points to an ORDFS server
2. A TXT record at `_ordfs.{domain}` contains `ordfs={inscriptionId}`
3. When a browser requests the domain, ORDFS reads the TXT record, fetches the inscription from the blockchain, and serves it as HTTP

For Bitcoin OS this means: Each application domain (e.g., `bitcoin-writer.app`) resolves to an ORDFS server, which serves the application code directly from an on-chain inscription. The DNS record is the only off-chain dependency.

5.2 React On Chain

React On Chain is a deployment pattern for inscribing complete React applications on Bitcoin. The process:

1. **Build** the application as static HTML/JS/CSS (via `next export`, Vite, or any bundler)
2. **Analyse** the dependency graph — identify shared libraries, assets, and modules
3. **Inscribe** dependencies in topological order (leaves first), each as a separate ordinal
4. **Rewrite** internal references to point to on-chain inscription IDs instead of relative paths
5. **Inscribe** the root `index.html` as the final ordinal, referencing all dependencies

Recursive inscriptions are the key capability: an HTML inscription can reference JavaScript inscriptions by their on-chain ID. When ORDFS serves the HTML, it resolves these references, serving the composed application as if it were a traditional web deployment.

Shared dependencies (React 19, Tailwind CSS, `@bitcoin-os/kernel`) are inscribed once. All 50 applications reference the same shared ordinals. This dramatically reduces per-app inscription size — a typical application's unique code is 50-200KB after shared deps are factored out.

5.3 Versioning

Each inscription is immutable. Updating an application means inscribing a new version. A bootstrapper inscription (following the React On Chain pattern) acts as a version router:

1. The bootstrapper is the stable entry point (pointed to by DNS)
2. It queries indexed metadata for the latest version inscription
3. It redirects the browser to the latest version's content
4. Previous versions remain permanently accessible by their inscription IDs

`BitGit` (`npm i -g bitgit`) automates this: `bit push` from an app directory builds, inscribes, and updates the version pointer in a single command.

5.4 Hybrid Architecture

Not all application logic can run on-chain. Database queries, real-time websockets, email relay, and third-party API integrations require server-side compute. The architecture accommodates this:

- **Frontend**: Inscribed on Bitcoin, served via ORDFS (immutable, verifiable)
- **API layer**: Runs on traditional infrastructure (Vercel, Hetzner) for dynamic operations

- **Blockchain**: Handles identity, payments, file storage, and access control

The frontend calls the API layer for dynamic data, but the frontend itself is permanent and uncensorable. If the API layer goes down, the application degrades gracefully — static content and on-chain operations continue to function.

---

6. Layer 3: Identity (Path 401)

6.1 Protocol Overview

Path 401 implements self-sovereign identity on the Bitcoin blockchain. Named after HTTP status code 401 (Unauthorized), it provides the authentication layer for Bitcoin OS.

An identity consists of:

- A **root inscription** establishing the identity on-chain
- **Strand inscriptions** for each linked authentication provider (OAuth)
- A **strength score** (1-4+) based on the number and type of linked providers
- A **cryptographic keypair** for signing and verification

6.2 OS Integration

When Bitcoin OS boots:

1. The System Bus checks for an existing $401 session in its internal state
2. If no session exists, the OS presents the $401 enrollment flow
3. The user authenticates via one or more OAuth providers (Google, GitHub, Twitter, etc.)
4. Each provider creates a strand inscription linked to the root identity
5. The System Bus stores the authenticated session and makes it available to all apps

Single Sign-On is automatic. Every app calls `bOS.identity.whoami()` and receives the same identity. Bitcoin Writer, Bitcoin Email, Bitcoin Drive — all see the same user without independent authentication.

6.3 Identity Strength

Level	Requirements	Capabilities
1	Single OAuth provider	Basic app access
2	Two providers	File operations, messaging
3	Three providers	Payment operations, token minting
4+	Four+ providers or KYC	Securities ($403), high-value operations

Higher identity strength unlocks more system capabilities, creating a natural incentive for users to link additional providers.

---

7. Layer 4: Payments and Token Economy (Path 402)

7.1 Protocol Overview

Path 402 implements the payment layer, named after HTTP status code 402 (Payment Required). It is a BSV-21 token with 21 million supply, minted via Proof of Indexing (PoW20).

7.2 Unified Wallet

The current system has six competing wallet implementations. Path 402 integration through the System Bus replaces all of them with a single wallet manager:

1. **Connection**: The wallet manager connects to MetaNet Desktop (localhost:3321), Yours Wallet (browser extension), or a local keypair
2. **UTXO Management**: A single UTXO set shared across all apps
3. **Transaction Construction**: All apps submit transaction requests to the System Bus, which batches and broadcasts them
4. **Balance Tracking**: Real-time balance updates pushed to all connected apps

7.3 Per-App Token Economy

Each application can mint BRC-100 tokens through the System Bus:

- **Bitcoin Writer**: Mints tokens representing published articles (content provenance)
- **Bitcoin Music**: Mints tokens representing music rights and royalty shares
- **Bitcoin Drive**: Mints tokens representing storage allocation
- **Bitcoin Spreadsheet**: Mints tokens representing shared workbooks
- **Bitcoin Exchange**: Facilitates trading of all token types

The System Bus's $402 client handles the cryptographic operations: `PushDrop.lock()` for token creation, `createAction()` for transaction construction, and ARC broadcasting for network propagation.

7.4 Resource Pricing

On-chain operations have real costs (transaction fees, inscription fees). The $402 client provides transparent pricing:

```
Write 1KB to ORDFS:     ~0.001 BSV
Mint BRC-100 token:     ~0.0005 BSV
Transfer token:         ~0.0003 BSV
List on 1sat.market:    ~0.0002 BSV
Identity strand ($401): ~0.001 BSV
```

The System Bus aggregates operations where possible, batching multiple app requests into single transactions to minimise fees.

---

8. Layer 5: Access Control and Securities (Path 403)

8.1 Protocol Overview

Path 403 implements access control, named after HTTP status code 403 (Forbidden). It manages securities tokens and KYC-gated access, requiring $401 identity verification at Level 4+.

8.2 Application Tiers

Each application operates on a tiered access model:

Tier	Access Method	Features
Free	Static app from ORDFS, $401 Level 1+	Core functionality
Standard	$402 micropayment per operation	Extended features, API access
Premium	$403 token holder	Unlimited access, priority API
Enterprise	Custom $403 contract	White-label, SLA guarantees

8.3 Smart Contract Enforcement

Access control is enforced via sCrypt contracts. When a user attempts a premium operation:

1. The API checks the user's $401 identity
2. It queries the bsv21-overlay for $403 tokens held by that identity
3. If a valid token is found, the operation proceeds
4. If not, the user is prompted to acquire a $403 token (via $402 payment or 1sat.market purchase)

The $403 token itself is a BSV-21 token whose genesis output contains the access grant parameters — which app, which scope, what expiry.

8.4 Revenue Distribution

$403 token revenue flows through the $DIVVY distribution system:

```
$403 token purchase
    → Revenue collected in $402
    → $DIVVY smart contract distributes to:
        → $BOASE holders (studio)
        → $KINTSUGI holders (AI engine)
        → $NPG holders (company shareholders)
        → App-specific token holders
```

This creates a direct link between application usage and token holder returns.

---

9. The Exchange Layer (1sat.market)

9.1 OrdinalLock Contracts

The 1sat.market exchange operates via OrdinalLock — an sCrypt smart contract that creates trustless, non-custodial listings directly on the blockchain.

When a seller lists an ordinal:

1. The ordinal is spent into a UTXO locked with the OrdinalLock contract
2. The contract encodes the seller's public key hash and the required payment output
3. The listing UTXO is visible on-chain to any marketplace frontend

When a buyer purchases:

1. They construct a transaction spending the listing UTXO
2. The contract cryptographically enforces that the seller's payment output is included
3. The buyer receives the ordinal; the seller receives payment
4. No intermediary, no escrow, no platform dependency

The Bitcoin ledger is the order book. Any frontend can read listings, any wallet can execute purchases. This is the app store for Bitcoin OS — applications, media, tokens, and digital assets all trade through the same mechanism.

9.2 Partially Signed Transactions (PSBTs)

For cases where smart contracts are unnecessary, Bitcoin's native sighash flags enable atomic swaps:

- Seller signs their input with `SIGHASH_SINGLE | SIGHASH_ANYONECANPAY`
- This commits to their payment output but allows anyone to add inputs/outputs
- Buyer completes the transaction with their payment inputs and ordinal-receive output
- Both parties' interests are cryptographically guaranteed

PSBTs support Dutch auctions (declining prices) and bidding natively.

9.3 Integration with Bitcoin OS

The System Bus's exchange client exposes these operations through the standard API. Any app can:

- **List**: Bitcoin Art creates a painting → `bOS.exchange.list(paintingId, price)`
- **Browse**: Bitcoin Apps Store queries listings → `bOS.exchange.listings({ category: 'apps' })`
- **Purchase**: User buys an app → `bOS.exchange.buy(listingUtxo)`
- **Cancel**: Creator delists → `bOS.exchange.cancel(listingUtxo)`

---

10. Indexing Infrastructure

10.1 The Indexing Stack

On-chain data must be indexed to be queryable. The indexing infrastructure has four layers:

JungleBus (GorillaPool) — A subscription-based blockchain data feed. It indexes every BSV transaction and serves filtered streams to subscribers. Applications subscribe with a filter and receive real-time matching transactions.

1sat-indexer — A Go service that subscribes to JungleBus and maintains a PostgreSQL + Redis index of all ordinal inscriptions. It tracks origins, transfers, and metadata.

bsv21-overlay — A specialised Go service for BSV-21 fungible token tracking. It maintains token balances, transfer histories, and UTXO sets per token per address.

GorillaPool REST API — The public query interface at `ordinals.gorillapool.io/api/`. Returns inscription metadata, UTXO sets, and token balances.

10.2 Bitcoin OS Indexing Requirements

The System Bus's ORDFS and exchange clients query the indexing infrastructure for:

- **File lookups**: `bOS.fs.read(id)` → GorillaPool API → inscription content

- **Token balances**: `bOS.pay.balance()` → bsv21-overlay → token UTXOs
- **Listings**: `bOS.exchange.listings()` → 1sat-indexer → OrdinalLock UTXOs
- **Identity verification**: `bOS.identity.verify()` → 1sat-indexer → $401 chain

For production deployment, Bitcoin OS can either rely on GorillaPool's public infrastructure or self-host the indexing stack (1sat-indexer + bsv21-overlay + PostgreSQL + Redis) for full sovereignty.

---

11. Smart Contract Integration (sCrypt)

11.1 Overview

sCrypt is a TypeScript-embedded DSL that compiles to native Bitcoin Script. In Bitcoin OS, smart contracts serve three purposes:

1. **Exchange**: OrdinalLock contracts for trustless trading
2. **Access Control**: $403 token contracts gating premium features
3. **Application Logic**: Per-app contracts for domain-specific operations

11.2 The UTXO Model

Each contract is a UTXO with a compiled script as its locking condition. To execute a contract method, a transaction is constructed that spends the UTXO with an unlocking script containing the method arguments. If the script evaluates to true, the spend is valid.

This model maps naturally to Bitcoin OS operations:

- **Minting a token** creates a UTXO with a PushDrop locking script
- **Transferring a token** spends the UTXO and creates new ones
- **Listing for sale** creates a UTXO with an OrdinalLock script
- **Purchasing** spends the OrdinalLock UTXO with a valid payment

11.3 Ordinal Compatibility

The inscription envelope (`OP_FALSE OP_IF ... OP_ENDIF`) is inert — `OP_FALSE` ensures the IF branch never executes. This means any sCrypt contract can be combined with an ordinal inscription. A single UTXO can simultaneously be:

- A 1-satoshi ordinal (carrying an inscription)
- A BSV-21 token (with deploy+mint metadata)
- A smart contract (with sCrypt locking logic)

This composability is fundamental to Bitcoin OS: a document in Bitcoin Writer is simultaneously a file (inscription), an asset (token), and a tradeable good (OrdinalLock-compatible).

---

12. Migration Path

Phase 1: The System Bus (Weeks 1-3)

Build `@bitcoin-os/kernel` as a SharedWorker with the typed API described in Section 4. Unify the six existing wallet implementations into a single wallet manager within the bus. Replace the current 3-message postMessage bridge with the full System Bus protocol.

Deliverable: Any app that imports `@bitcoin-os/kernel` immediately gains shared identity, wallet, and IPC.

Phase 2: Identity Integration (Weeks 3-5)

Integrate Path 401 as the OS login, replacing the current HandCash/multi-provider modal. The System Bus's identity manager handles enrollment, session management, and cross-app SSO.

Deliverable: One login for all 50 apps. Identity anchored on-chain.

Phase 3: Shared Dependencies On-Chain (Week 5-6)

Inscribe React 19, Tailwind CSS, and `@bitcoin-os/kernel` as base ordinals on Bitcoin. These become the shared runtime referenced by all applications via recursive inscriptions.

Deliverable: Common libraries permanently available on-chain, reducing per-app inscription size to unique code only.

Phase 4: First Wave On-Chain Apps (Weeks 6-8)

Convert the five simplest applications to static builds and inscribe them via BitGit. Candidates: Bitcoin Paint, Bitcoin Art, Bitcoin 3D, Bitcoin Calendar, Bitcoin Education. These apps have minimal API dependencies and can function entirely client-side.

Deliverable: Five applications running from Bitcoin inscriptions via ORDFS.

Phase 5: Payment Integration (Weeks 8-10)

Wire Path 402 payments through the System Bus for all file and token operations. Every `bOS.fs.write()` and `bOS.pay.mintToken()` call flows through the unified $402 client.

Deliverable: On-chain operations (inscriptions, tokens, transfers) available to all apps through a single payment interface.

Phase 6: Remaining Apps On-Chain (Weeks 10-14)

Convert all remaining applications to the hybrid architecture: static frontends inscribed on Bitcoin, API layers on traditional infrastructure for dynamic operations.

Deliverable: All 50 applications with on-chain frontends.

Phase 7: Exchange Integration (Weeks 14-16)

Add 1sat.market exchange operations to the System Bus. Applications can list, buy, sell, and auction digital assets through the standard API.

Deliverable: Every app can participate in the on-chain marketplace.

Phase 8: Access Control (Weeks 16-18)

Implement Path 403 access control with sCrypt contracts. Premium features gated by token ownership, revenue flowing through $DIVVY.

Deliverable: Tiered access model with on-chain enforcement.

Phase 9: The OS On-Chain (Week 18-19)

Inscribe the Bitcoin OS shell itself via React On Chain. The bootstrapper inscription becomes the permanent entry point, served via ORDFS at `bitcoin-os.website`.

Deliverable: The complete operating system — shell, apps, identity, payments, and access control — running from Bitcoin.

---

13. Security Considerations

13.1 Key Management

The System Bus holds the user's cryptographic keys in a SharedWorker — isolated from all application code. Applications request operations (sign, pay, inscribe) but never receive raw keys. This is analogous to a hardware security module: the keys never leave the trusted boundary.

13.2 Application Sandboxing

Each application runs in its own iframe with standard browser sandboxing. The System Bus mediates all cross-app communication. An application cannot:

- Access another application's DOM
- Read another application's cookies or localStorage
- Directly call the BSV SDK
- Construct transactions without System Bus approval

13.3 Permission Model

The $403 access control extends to inter-app communication. An application must hold the appropriate $403 token to:

- Read files created by another application
- Send messages to another application
- Access premium API endpoints
- Mint tokens on behalf of the user

13.4 Immutability

Inscribed applications are immutable. A malicious update requires a new inscription with a new ID. The version router (bootstrapper) is the only mutable reference — and its update history is itself recorded on-chain, providing a complete audit trail.

13.5 Censorship Resistance

The application code exists on every Bitcoin node that has indexed the relevant blocks. To censor an application, an attacker would need to:

1. Remove the inscription from the Bitcoin blockchain (impossible)
2. Shut down all ORDFS servers (anyone can run one)
3. Remove DNS records (mitigated by multiple registrars and on-chain DNS via $bDNS)

The system degrades gracefully: even if all ORDFS infrastructure becomes unavailable, the inscriptions remain permanently on-chain, retrievable by anyone with a Bitcoin node.

---

14. Conclusion

Bitcoin OS transforms a collection of 50 independent web applications into a unified operating system by recognising that Bitcoin already provides all the primitives an OS requires: an immutable file system (inscriptions), a native payment layer (transactions), a programmable access control system (Script), and a globally replicated state machine (the blockchain).

The Path 401/402/403 protocol stack maps HTTP status codes to OS services — authentication, payment, and authorisation — creating an intuitive mental model for developers. The System Bus (SharedWorker) provides the kernel-level mediation that connects applications to these services. ORDFS and React On Chain provide the deployment infrastructure that makes applications permanent and uncensorable.

The end state is an operating system that boots from an inscription, authenticates via on-chain identity, pays for resources with native tokens, controls access through smart contracts, and trades assets on a trustless exchange — all on Bitcoin.

References

1. Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System.
2. 1Sat Ordinals Protocol Specification. docs.1satordinals.com
3. BSV-20 Token Standard. docs.1satordinals.com/fungible-tokens/bsv20
4. BSV-21 Token Standard. docs.1satordinals.com/fungible-tokens/bsv-21
5. sCrypt Documentation. docs.scrypt.io
6. ORDFS Server. github.com/shruggr/ordfs-server
7. React On Chain. app.reactonchain.com
8. JungleBus. junglebus.gorillapool.io
9. 1sat-indexer. github.com/shruggr/1sat-indexer
10. BSV-21 Overlay. github.com/b-open-io/bsv21-overlay
11. OrdinalLock Tutorial. docs.scrypt.io/tokens/tutorials/ordinal-lock
12. js-1sat-ord Library. github.com/BitcoinSchema/js-1sat-ord
13. Path 401 Identity Protocol. path401.com
14. Path 402 Payment Protocol. path402.com
15. BitGit CLI. npmjs.com/package/bitgit
16. Bitcoin Apps Suite. github.com/bitcoin-apps-suite
17. Bitcoin OS. github.com/bitcoin-corp/bitcoin-OS

Document Version: 1.0 **Date:** 19 February 2026 **Licence:** Open-BSV-4.0 **Contact:** info@thebitcoincorporation.com

THE BITCOIN CORPORATION LTD UK Company No. 16735102