TECHNICAL WHITEPAPER

uOS

A post-quantum, end-to-end encrypted workspace for people and organisations

Version 4.3 · September 2026 · Alpha

UltimaOS

Contents

Executive summary

uOS is a complete workspace — chat, mail, video meetings, files, documents, spreadsheets, presentations, boards, calendar, notes, tasks, AI assistants, shift planning and an app studio — delivered in the browser, on one identity and one security model. Its premise is simple: **the operator of a service should not be able to read what its users create.** In uOS, content is encrypted on the user's device, with keys the user alone holds, before it reaches our servers.

Three properties set uOS apart:

- **Post-quantum cryptography, end to end.** Current account keys, every per-recipient content key, every uOS session and the media of every call are protected by NIST's standardised post-quantum algorithms (ML-KEM and ML-DSA) *combined* with proven classical ones (X25519 and Ed25519). Content captured today cannot be decrypted by a quantum computer tomorrow, and a flaw in either family alone does not break the system. Since September 2026 the service runs the stronger Category 5 suite (ML-KEM-1024, ML-DSA-87) for new keys. Connections to uos.sh negotiate the hybrid TLS group `X25519MLKEM768` with current browsers. The exceptions — legacy accounts until their next login, browsers without hybrid TLS, and the classical primitives listed in Appendix A — are stated in Section 16.
- **One encryption model across a whole workspace.** Where other privacy products protect a mail inbox or a document editor, uOS applies the same envelope to messages, files, documents, calendars, AI conversations, boards, organisation feeds and staff rosters — and publishes, per application, exactly what remains visible to the server.
- **Built for organisations, not only individuals.** Organisation workspaces add roles, owner-quorum governance, signed compliance profiles, litigation hold, retention rules, mail egress control and a *key escrow* that lets an organisation recover its own business records — without giving the operator any access, and without touching members' private workspaces.

This paper is written for two readers. Decision-makers will find the product, the trust model and the business principles in Parts I and IV. Security reviewers will find the architecture, the cryptographic protocols, a per-application coverage matrix and a threat model in Parts II and III, with the exact primitives and parameters in the appendices. Where a guarantee has limits, the limit is stated next to the guarantee.

PART I — THE CASE FOR UOS

1. The problem: trust by policy

The software most people and companies rely on — office suites, cloud drives, team chat, mail — is built on a common arrangement: the provider runs the code, stores the data, and holds the keys. Encryption "at rest" protects against a stolen disk, not against the provider. Everything else rests on policy: privacy statements, contracts and certifications.

Policies are made in good faith, but they are not a technical boundary. They can change with a new owner, a new business model or a new law. They can be overridden by a court order in another jurisdiction. They cannot stop an insider with production access, and they do not survive a breach. For regulated organisations the consequence is concrete: every sensitive file stored in a mainstream cloud is, in principle, readable by someone outside the organisation.

A second pressure is arriving on a fixed schedule. Public-key algorithms in use today — RSA, elliptic curves — will be broken by a sufficiently large quantum computer. Adversaries do not need to wait: traffic and stored ciphertext recorded now can be decrypted later ("harvest now, decrypt later"). Security agencies in Europe and the United States have set timelines for migrating to post-quantum cryptography; data with a long confidentiality life needs that protection already.

2. The uOS approach

uOS replaces trust by policy with trust by construction:

1. **Encrypt before upload.** Content is encrypted in the user's browser. The server stores and relays ciphertext it cannot open, because it never receives the keys.
2. **Keys belong to people.** An account's private keys are generated on the user's device from random seed material and unlocked from secrets only the user knows. The operator cannot reset, recover or impersonate an account.
3. **Minimise the rest.** What the server must see to function — routing, ordering, timing — is reduced wherever the product allows: sender identities are sealed, file sizes are padded, timestamps coarsened, presence is short-lived, and no tracking, advertising or analytics runs anywhere in the product.
4. **Say exactly what is and is not covered.** A precise, published boundary is more useful to a reviewer than a broad promise. Section 9 lists, per application, what is encrypted and what the server can see.

3. uOS at a glance

uOS runs in any modern browser on desktop and mobile, as a desktop-style workspace with windows or as a full-screen mobile layout. It is available in 28 European languages.

Area	Applications
Communication	uChat (direct, group, channel and confidential chats), uMail, uMeet (video meetings), Contacts, uLine (a feed inside each workspace)

Area	Applications
Productivity	Files, uDoc, uSheet, uPresent, Boards (kanban), Calendar, uNotes, uTodo, Projects
Organisation	uShift (shift planning, absences, time tracking), Admin, platform settings and compliance
Intelligence	uAI (assistant with tools), uSages (conversations with historical personas), uAssistant (built-in help), Desktop Agent
Build & explore	Studio (build and publish apps), Marketplace, uBrowse and uNews, the uOS Guide

Platforms. A person has one *core account* — their cryptographic identity — and can belong to several *platforms*. A platform is an isolated workspace with its own members, roles, apps, branding and settings. There are four kinds: the personal core space, organisation platforms (for-profit and non-profit), and neutral *collaboration* platforms for work across organisations. A person's private workspace and their organisation's workspace are separate by design, including in who can ever recover the content (Section 10.4).

PART II — ARCHITECTURE AND CRYPTOGRAPHY

4. System architecture

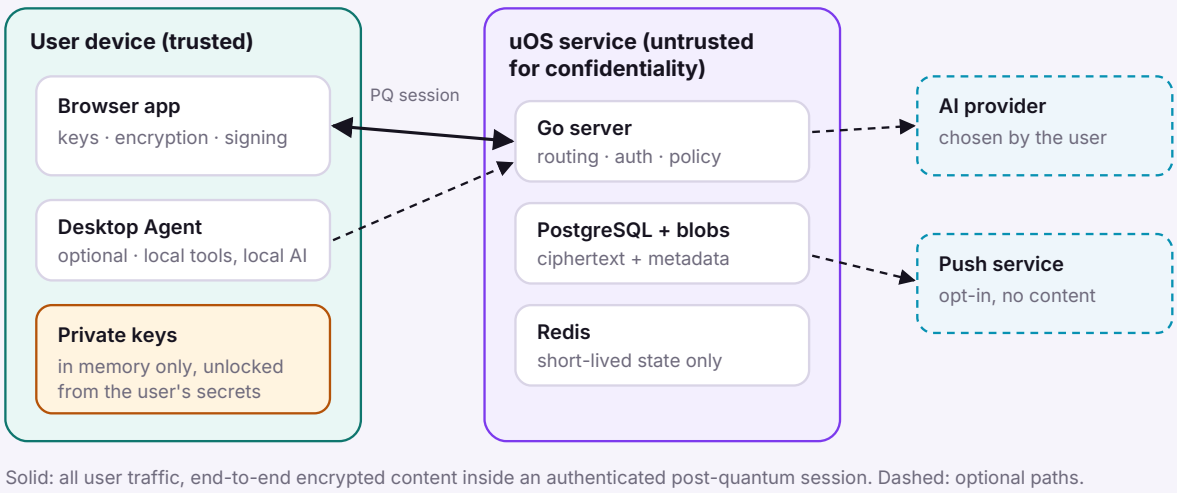

Figure 1 — Encryption and key custody sit on the user's device. The service stores and relays ciphertext.

Client. An Angular single-page application that performs all key handling and encryption in the browser. Account private keys live in memory for the duration of a session and are never written to browser storage. The only persistent browser storage is a device vault holding the forward-secrecy keys of that device (Section 8.2), encrypted under a non-extractable browser key; other browser storage is cleared on every load. Application data is kept on the server — encrypted — so that a user can sign in from any device.

Server. A Go service that authenticates sessions, routes messages, enforces platform membership and organisation policy, and stores data. It is treated as *untrusted for confidentiality*: the system is designed so that a fully compromised server yields ciphertext and metadata, not content.

Storage. PostgreSQL holds ciphertext, public keys and operational metadata; encrypted file bodies are stored as opaque blobs. Redis holds short-lived state only — handshake challenges, presence, rate-limit counters. Session tokens and signing keys are held in server memory, never in Redis or on disk; a restart ends every session.

Transport. Everything a user does travels over one authenticated WebSocket (`/ws`), established with a post-quantum handshake (Section 6.1). Where a network blocks WebSockets, the client falls back to an HTTPS long-poll channel that runs the same handshake and carries the same encrypted, signed frames. The remaining HTTP surface is small and listed in Appendix C: login and key-derivation lookup, PIN checks, account rotation transfers, ticket-gated bulk upload of ciphertext, public invite and platform pages, and published Studio sites. The requests among these that carry anything sensitive — the login lookup and its answer, unlock, PIN checks, change of secrets and rotation — are themselves sealed with the hybrid KEM (ML-KEM-1024 + X25519) to a server key signed by the pinned server identity, so they do not depend on TLS either.

Desktop Agent. An optional program on the user's computer that connects *outbound* to the service, so no ports are opened. It lets the AI assistant work with local files and commands under explicit permissions, and can run AI models locally.

5. Identity and accounts

5.1 Registration and unlock

A uOS account is a set of key pairs, not a username and password on a server. At registration the browser generates random seed material — a 32-byte ML-DSA seed and a 64-byte ML-KEM seed — with a cryptographic random number generator. From this *seed bundle* it derives the account's post-quantum and classical signing and encryption key pairs, using domain-separated key derivation for each.

To make the account usable from any device, the seed bundle is stored on the server in wrapped form. The user chooses two secrets — **Secret 1** (at least 10 characters, mixed case) and **Secret 2**, a pass-phrase (at least 20 characters, mixed case plus a digit or symbol) — and an account identifier in e-mail form. The browser derives a key-encryption key with Argon2id and wraps the seed bundle with it:

```
KEK     = Argon2id(NFKC(S1) ‖ 0 ‖ NFKC(S2) ‖ 0 ‖ NFKC(identifier) ‖ 0, salt₃₂, m = 256 MiB, t =
3, p = 2)
wrapped = XChaCha20-Poly1305(KEK, nonce₂₄, seed bundle, AAD = account id ‖ KDF version ‖ secret
epoch)
```

Each input is Unicode-normalised (NFKC) and trimmed before derivation, so the same secrets unlock the account on any operating system or keyboard; the fields are joined with zero-byte separators. The server stores the wrapped bundle, the salt, the public keys and a peppered HMAC of the identifier — never the secrets, a password verifier, or the plaintext identifier. On login, the browser fetches the candidate wrapped bundle, derives the KEK locally and unwraps. An unknown identifier receives a decoy of the same shape, derived deterministically from the identifier and a server secret, so repeated lookups return the same answer and do not reveal whether an account exists.

The wrapped bundle is released only to someone who proves the secrets. The lookup returns only the account's Argon2id salt. From the Argon2id output the browser derives, besides the KEK, a separate login key `auth_key = HKDF-SHA256(Argon2id output, "uOS/login-auth/v1" ‖ account id)`; the server stores only `SHA-256("uOS/login-verifier/v1" ‖ auth_key)` and releases the wrapped bundle, through a request sealed with the hybrid KEM, only when the presented login key matches. Wrong keys, unknown accounts and decoys receive the same refusal, and attempts are limited per account and address (5 per minute) and per account overall (30 per minute), so a stranger cannot lock the owner out. Knowing an identifier therefore no longer yields an offline-guessing target: offline guessing needs the server's database. The mechanism uses only symmetric primitives, so it adds no quantum-vulnerable assumption. It is not OPAQUE: the server sees the login key during a login. Accounts switch to this gate automatically at their next login.

Consequences of this design:

- The operator **cannot reset or recover** an account. There is no "forgot password" link; recovery is in the user's hands (Section 11).

- Anyone holding a copy of the database can still guess offline: each guess of the two secrets costs a 256 MiB Argon2id evaluation. New secrets follow NIST SP 800-63B: Secret 1 has at least 12 characters and Secret 2 at least 20, the two differ from each other and from the identifier, and there are no composition rules. The browser refuses any secret found in a list of the million most common breached passwords (SecLists, MIT licence), shipped as a Bloom filter and checked entirely on the device, including without spaces and without trailing digits or symbols; nothing about the secret is sent anywhere. Existing accounts are advised, not forced, to strengthen short secrets.
- Private keys exist only in the memory of an unlocked session.

5.2 PIN and high-trust actions

A separate PIN, verified on the server against an Argon2id hash, gates sensitive actions — such as unlocking a returning session, changing secrets and rotating the identity — in addition to the session's signatures.

5.3 Rotation and changing secrets

Users can change their secrets or rotate their entire identity. A rotation creates a new key epoch, rewraps every content key the account can open for the new key, and re-points every reference to the old key, in one transaction that can be resumed for 24 hours. A mandatory PIN, a recovery-share gate and an automated coverage check (Section 14.3) protect the process. Changing secrets replaces the wrapped bundle and invalidates old secrets and old recovery material.

Accounts created before the random-seed model, whose keys follow directly from their secrets, are **upgraded at their next login**. Before the workspace opens, a step that cannot be skipped asks for the PIN and runs a rotation to a fresh random seed bundle in the current suite. The wrapped bundle, the login verifier, the key-transparency entries and every content key are replaced in one commit, and nothing changes if the step fails. An operator can additionally set a date after which such an account can do nothing but this upgrade.

5.4 Server identity

Clients authenticate the server, not only the reverse. The server's post-quantum signing keys are **pinned** in the client: during every handshake the server signs the client's challenge, and the client refuses any server key whose fingerprint is not pinned. Every frame the server sends afterwards is signed and verified. A network attacker who breaks or terminates TLS therefore cannot impersonate the uOS server *to a client that is already running intact code*. The pins, however, are part of the web client, and the web client is itself delivered over TLS: whoever controls TLS at the moment the app is loaded can serve different code, including different pins. The pins protect a client obtained intact — a cached release, a verified build or a native app — and do not by themselves protect against malicious code delivery (Section 13.2).

5.5 Contacts, key changes and verification

When a contact rotates their keys, the server fans out a key-change record signed by *both* the contact's old and new keys, and clients warn before sending to the new key; a server cannot fabricate a rotation of a key it does not hold. First contact is different: when Alice messages a new contact or shares a file with Bob, her client encrypts to the key the server serves for Bob. A malicious server could serve a key of its own for any contact whose key has not been verified. **Key transparency** closes most of this gap. Every account key epoch (registration, rotation, suite upgrade) is appended, in the same database transaction,

to an append-only RFC 6962 Merkle log whose tree heads the server signs with its pinned identity key. Before encrypting to a contact key, a client requires an inclusion proof that the key is that account's latest entry, rejects key changes the log does not show, and checks every tree head for consistency with every earlier head it has seen — including heads its contacts report inside encrypted chat envelopes. At each login a client audits its own account: every entry must be its current key or linked to it by key-change records signed by both keys, and from the last head it verified (kept, signed, in its encrypted settings) it rederives the root over every entry added since, so a key the server registers for an account is shown to its owner. This holds on every encryption path by construction. The client's encryption functions accept only a recipient obtained through the verified lookup, the session's own keys, keys the browser has just generated, or a short list of named non-account keys, and a build-time check rejects any other use. The paths are chat, attachments, mail, files, documents, spreadsheets, presentations, projects, boards, calendar, meetings, uLine, uShift, profile shares, escrow share holders and rotation. A device prekey is used only when its device is signed by an account key proven in the log, and the device has signed the prekey. A call's offer and answer are accepted only from a proven identity key. When a recipient's key cannot be proven, group operations leave that recipient out and say so; operations that would otherwise lose someone's access, such as escrow shares and re-keys, stop instead. Organisation escrow keys are in the log too, as statements signed by the owner who set them (Section 10.4). The log cannot prove that a key is absent and does not bind accounts to people: a server that shows different logs to different users is detected only when their views meet, and a newly invented account impersonating a person still requires comparing a symmetric *safety number* out of band.

6. Cryptography

6.1 Hybrid by principle

uOS combines a post-quantum and a classical algorithm for every asymmetric operation. Both must succeed; neither is accepted alone:

- **Key establishment:** ML-KEM (FIPS 203) with X25519. The shared secret is derived with HKDF over both secrets and both ciphertexts.
- **Signatures:** ML-DSA (FIPS 204) with Ed25519. Both signatures must verify.
- **Symmetric encryption:** XChaCha20-Poly1305 with 256-bit keys and random 192-bit nonces, which remain strong against quantum attack.

Hybrid construction is the approach recommended by European security agencies (ANSSI; BSI TR-02102-1, which also recommends ML-KEM-1024) during the post-quantum transition: an unexpected weakness in the young post-quantum algorithms is caught by the classical half, and quantum attacks on the classical half are caught by the post-quantum one. TLS protects the connection as well. Once the app is running, the confidentiality of content and of sensitive requests does not depend on TLS; the delivery of the app itself does (Section 13.2), which is why uos.sh also negotiates hybrid post-quantum TLS (`X25519MLKEM768`) with browsers that support it.

6.2 Cryptographic suites

Every key, ciphertext and signature belongs to a **suite**. A deployment accepts a fixed set of suites and writes exactly one; nothing is negotiated, so there is nothing to downgrade.

Suite	Key encapsulation	Signatures	AEAD	KDF hash	Use
v2	ML-KEM-768 + X25519	ML-DSA-65 + Ed25519	XChaCha20-Poly1305	SHA-256	accepted; existing keys
v3-hybrid	ML-KEM-1024 + X25519	ML-DSA-87 + Ed25519	XChaCha20-Poly1305	SHA-384	**written by uos.sh since September 2026**
v3-cnsa	ML-KEM-1024	ML-DSA-87	AES-256-GCM	SHA-384	planned, dedicated deployments only

The v3-hybrid values are *composite*: the classical half is carried inside every public key, ciphertext and signature, so no field can exist without it, and stripping it yields a malformed value rather than a weaker one. The construction is uOS's own. A composite signature is an ML-DSA-87 signature with a suite-specific context string followed by an Ed25519 signature over a suite-prefixed message, so neither half verifies as a signature of another protocol using the same key. The KEM follows the design of X-Wing — the classical ciphertext and public key are bound into the combiner — but uses HKDF-SHA384 and is not wire-compatible with X-Wing; the signatures do not claim conformance with the IETF LAMPS composite drafts. Alignment with those standards is on the roadmap. Every key-derivation label is bound to its suite, so a ciphertext cannot be opened under another suite's derivation. New accounts, new keys and everything the server signs use v3-hybrid; existing accounts move to it with one action in Settings, from the same seed bundle and without new secrets. The *v3-cnsa* suite targets customers who must follow the US CNSA 2.0 profile, on their own or dedicated infrastructure only.

Shared known-answer test vectors pin every primitive and encoding across implementations: Go generates them, the TypeScript client reproduces every deterministic value byte for byte, and the Dart mobile code checks the v2 vectors.

6.3 Session handshake

1. The client opens `/ws` and sends its identity key.
2. The server answers with a random challenge, its identity key and a signature over the challenge and the client's key. The client checks the key against its pins and verifies the signature.
3. The client signs the challenge and the server's key, and sends key-encapsulation material for the hybrid KEM in its suite.
4. Both sides derive a session key with domain-separated HKDF. From then on every frame is encrypted and signed, and carries a sequence number.

A v2 identity receives the v2 handshake; a v3 identity receives the server's v3 key and the composite KEM. Handshakes are rate-limited per address and per account.

7. Protecting content

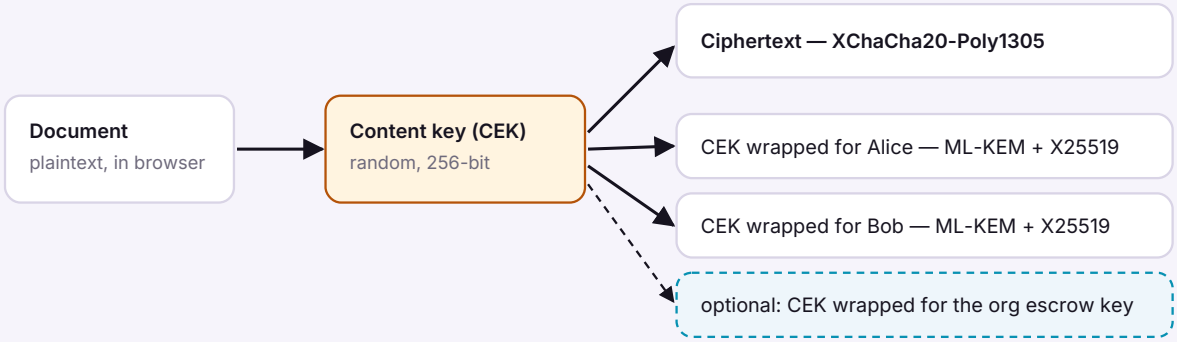

Figure 2 — Content is encrypted once; its key is wrapped separately for each reader. Adding a reader wraps the key again; removing one deletes their wrap. The server stores only the right-hand column.

7.1 Envelope encryption

Content in uOS is encrypted under a random content key, and that key is *wrapped* — encapsulated with the hybrid KEM — for every person allowed to read it. Four patterns cover the product:

- **Per-resource keys** for files, documents, spreadsheets, presentations, boards, projects and calendar entries. Sharing wraps the key for the new reader; revoking deletes the wrap, and the server enforces the permission on download. Revoking a reader **re-keys** the resource: the owner's client generates a new content key, re-encrypts the current content (for files a new blob, for boards every field, comment and attachment), wraps it for exactly the remaining readers — and the escrow key where applicable — and the server applies revocation, ciphertext and wraps in one transaction that increments a per-resource key version. Saves and live-collaboration frames under an old key version are refused. A removed reader can still read copies obtained before the revocation, but nothing saved afterwards. Board fields are bound to their board and field with associated data, so ciphertext cannot be moved between records.
- **Per-recipient copies** for chat, mail and notifications, where each recipient receives an individually encrypted copy.
- **Workspace keys** for content meant for every member of a platform, such as the uLine feed. Browsers create and distribute each key *epoch* to members, and rotate it when a former member still holds the current one.
- **Scope keys** in uShift, where the staff, HR and all-members scopes each have their own epoch keys, and personal records are sealed to the scope and to each person involved.

Live collaboration in uDoc and uSheet encrypts every update and awareness message under the document key with fresh nonces; the server rejects plaintext collaboration frames.

7.2 Large files

Large files are uploaded as ciphertext over a ticket-gated HTTP path: the client obtains a server-signed, short-lived ticket over the WebSocket and uploads the encrypted file with it. File sizes are padded to power-of-two buckets and file timestamps are coarsened to the hour, so storage metadata reveals less about the contents.

8. Messaging privacy

8.1 Sealed sender

In uChat, the sender's identity travels *inside* each recipient's encrypted envelope. The message table stores no sender. Delivery is authorised with single-use tokens obtained through RFC 9474 RSA blind signatures (RSABSSA-SHA384-PSS-Randomized, 3072-bit dedicated key), so the server can check that a token is valid and unspent without being able to link it to the account that obtained it. Tokens carry a server-assigned, hour-aligned expiry of at most 12 hours; spent-token records, kept only to prevent replay, are deleted after 24 hours. Blocking and abuse reporting work on top of this model. Notifications do not record who contacted whom — only contact requests and join notices store a sender, because that relation is what they are about — a database trigger enforces this for every writer, and notifications are deleted 30 days after creation or 7 days after being read. Server logs of message delivery name neither the sender together with the conversation.

Sealed sender removes the stored sender graph; it does not hide who is connected at a given moment. The server still sees which authenticated session submits a message and when, and it knows the members of each conversation. Section 13.2 lists what remains visible.

8.2 Forward secrecy

Each of a user's messaging devices enrols its own key, signed by the account, and publishes signed, expiring prekeys (ML-KEM + X25519). Messages are encrypted to the recipient's device prekeys, so the compromise of an account key later does not expose messages in transit. To keep history readable on all of a user's devices, a device stores a copy of each received message on the server, re-encrypted to the account key. Stored history is therefore protected by the account key, not by forward secrecy: anyone who unlocks the account — for example by guessing its secrets from the wrapped bundle (Section 5.1) — can read the whole stored history. Forward secrecy protects messages in delivery. A shield badge in the chat shows when forward secrecy is active.

8.3 Chat classes

Organisation platforms distinguish **direct**, **group**, **channel** and **confidential** chats. Channels can only be created by administrators and are marked as escrowable; confidential chats are marked as such, can never be escrowed, and cannot change their class once set.

8.4 Presence

Online status is held in memory with a 90-second lifetime; last-activity stamps expire after 24 hours and last-seen stamps after 30 days. Nothing about presence is kept in the database.

9. What is protected — per application

The table states, for each application, what is end-to-end encrypted and what the server can see. "Visible" does not mean published: it means the server needs it to operate and could disclose it if compromised or compelled.

Application	End-to-end encrypted	Visible to the server
uChat	Message text, attachments, thread titles, sender identity	Conversation membership, timestamps, sizes, chat class
uMail	Subject and body, per recipient	Sender, recipients, folder, read and star flags, timestamps
uMeet	Meeting title and description; every audio and video frame, under keys the participants agree with ML-KEM-1024 + X25519	Who calls whom and when; network addresses in the signed call set-up
Files	Contents, file and folder names	Folder structure, padded size, hourly timestamp, sharing relations
uDoc, uSheet, uPresent	Contents, titles and live edits	Owner, readers, timestamps
Boards	All fields of private boards	Structure and card order; public boards are readable by design
Calendar	Title, description, location, meeting proposals	Start and end times, recurrence, participants
uNotes, uTodo	Titles, bodies, folder and category names	Order, due dates, completion
Contacts	The contact list itself (keyed-hash lookups, encrypted entries)	Pending contact requests
uAI, uSages, uAssistant, Projects	Conversations, reasoning, tool results, attachments, project content	Timestamps, model chosen; content is sent to the chosen AI provider during a request (Section 12)
uLine	Posts, under the workspace key	Author, time, reply structure
uShift	Rosters, contracts, requests, wishes, time entries	Teams, roles, periods, request status, approval chain, reminder times
Settings	Preferences	Language, theme, colours, scale and last platform, needed before unlock
Profile	—	Name, username, bio, location and avatar, which users publish to others by choice
Studio	Values each app stores for a user	App code and manifests (by design: reviewed and served to others)
uBrowse, uNews	—	Public content fetched on request

uMail is internal to uOS. It delivers only between uOS accounts; it does not send mail to, or receive mail from, internet e-mail addresses, so no uMail message passes through SMTP. "External recipients" in an organisation's mail egress policy (Section 10.3) are uOS accounts outside that organisation's platform. The only e-mail the service sends is invitation and optional security notices (Section 15), composed by the server.

Notifications are sent as opaque wake-up signals; the device fetches and decrypts the content itself. Web Push is opt-in and carries no message content.

PART III — ORGANISATIONS, RECOVERY AND AI

10. Organisations

10.1 Membership and invitations

Every member joins through an invitation, and each invitation is signed by its creator; the invitation chain is recorded, so every member's path into a workspace is accountable. In organisation platforms only owners, administrators and HR can invite. Roles range from owner and superuser through administrator and moderator to member and guest, with per-platform permissions.

10.2 Governance

Critical changes, such as ownership transfers and litigation holds, can require a quorum of owners. Each approval is an ML-DSA signature by a distinct owner, verified by the server and kept with the action for later audit.

10.3 Compliance controls

An organisation's **compliance profile** is a signed, versioned document with an append-only history. It controls which applications members can use (enforced on the server, not only in the interface), view-only and download restrictions, watermarking in Files, retention rules executed by a reviewed deletion queue, and a mail egress policy that allows, warns about or blocks mail to recipients outside the organisation. A **litigation hold**, created and released by owner quorum, blocks deletion of covered content types. Organisations record their jurisdiction; German organisations attest to their works-council agreement.

10.4 Organisation key escrow

Organisations need to recover business records when an employee leaves or loses access; private individuals need to know that no one else can. uOS provides both, separately:

- An organisation can create an **escrow key**. It is generated in the browser of the owner who creates it — who therefore sees the complete private key at that moment — and its private half is split with Shamir secret sharing among designated owners and share holders, each share encrypted to its holder. Rotating the escrow key re-wraps every covered content key under a new key.
- For the content types the organisation selects — files, documents, spreadsheets, boards, calendar, notes, tasks and mail — the content key is additionally wrapped for the escrow key (Figure 2).
- Recovery requires a quorum of share holders in their own browsers. The operator holds no share and cannot take part.

Escrow exists only on organisation platforms. It is absent — not switched off — on personal and collaboration platforms, and it never covers direct, group or confidential chats. Members are told when escrow applies. Every file, document, spreadsheet, board, calendar entry, note, task and e-mail whose content key is wrapped for escrow carries a mark reading "Recoverable by your organization", in lists and in the open item. Create forms show "Will be recoverable" when the platform covers the type, and chat channels carry an "Escrowable" mark.

The escrow key is signed and logged. Every escrow key is published as a statement naming the platform, the key, the threshold, the share holders, the time and the hash of the previous statement. The statement is signed with the identity key of the owner who set the key and recorded in the key-transparency log, in the same transaction that stores the key. Before wrapping anything for escrow, a member's app requires three things:

- an inclusion proof that the statement is the platform's latest escrow entry;
- a signature that verifies;
- proof that the signing key is an account key in the log.

It never falls back to a key the server serves. At every login, owners' apps check the platform's whole chain of statements and show any change they did not make, naming who signed it. To obtain escrow copies for a key of its own, a server would have to create an owner account and sign with it, and both would appear in the log, where the real owners see them. Keys set before statements were introduced get no new escrow copies until an owner confirms them.

11. Recovery

Because the operator cannot recover accounts, users set up recovery themselves, and are asked to do so right after registration:

- **Shamir recovery shares.** The browser splits the account's secrets into *n* shares, of which any *k* reconstruct them — 2 of 3, 3 of 5 (recommended) or 4 of 7. Shares can be given to trusted people (encrypted to their keys), printed, stored as QR codes or kept in a password manager. The server stores metadata about shares, never the shares.
- **Recovery file.** A downloadable file with the shares and a copy of the wrapped seed bundle, so the account can be restored even if the server could no longer supply it. It can optionally be protected with a passphrase (Argon2id, 256 MiB). On import, the file is checked against the account's public key and rejected if anything does not match.

An account whose secrets and recovery material are both lost cannot be restored — by design. A service that can restore your identity for you can also impersonate you.

12. Artificial intelligence

uOS includes AI across the workspace: the uAI assistant with tools for calendar, files and documents; uSages personas; the uAssistant help; an assistant panel in every app; and the Desktop Agent for local work.

AI conversations, reasoning traces, tool results and attachments are **stored end-to-end encrypted** to the user's key. Generating an answer, however, requires sending the relevant context to a language model. For that request, the server relays the context in memory without storing or logging it, and the **provider the user or organisation has chosen** processes it in plaintext. That the relay neither stores nor logs is a property of the server software — a policy, not a cryptographic guarantee: a compromised server could record the context of AI requests. Running a model locally through the Desktop Agent avoids the relay en-

tirely. Providers are configurable per platform and per user — hosted providers in the US, the EU and elsewhere, a custom endpoint, or a model running **locally** through the Desktop Agent, in which case nothing leaves the user's machine. The interface discloses which provider receives a request.

The Desktop Agent executes local actions only within the paths and command rules the user allows, with confirmation for sensitive operations. It holds its own device key, pairs with a one-time token, and verifies every frame the server sends. At pairing it pins its owner's account identity key: the pairing code the user copies from the browser carries a fingerprint of that key, so a key substituted by the server is rejected. Tools that write, delete or execute — shell commands, file creation and edits, fetches to local-network addresses — run only after the owner approves the exact action in the browser; the browser signs the approval (tool, working directory, every argument, identifiers, decision, time) with the account identity key, and the agent verifies it against the pinned key, rejecting stale, replayed, altered or unsigned approvals. Changes of the working directory, and requests to open a web address in the local browser, are signed the same way, and the pin follows a key rotation only through records signed by both keys. The server relays approvals but cannot create them. Agent updates are installed only when their manifest is signed with the uOS release key (ML-DSA-87 + Ed25519) compiled into the agent, and never downgrade. Read-only tools within the working directory still run without approval, and their results reach the server and the AI; credential directories (for example `~/.ssh`) are never read.

PART IV — ASSURANCE

13. Threat model

13.1 What uOS is designed to withstand

- **Compromise of the service.** A complete copy of the database and blob store yields ciphertext, public keys and metadata — no private keys and no content in the encrypted categories (Section 9).
- **Insiders.** An operator employee with full production access obtains the same.
- **Legal compulsion for stored data.** The operator can produce only what it holds: account metadata, platform membership, invitation records and the metadata in Section 9. It cannot produce encrypted content. Organisation escrow content can be recovered only by the organisation's share holders.
- **Network attackers** who observe or record traffic, including a future attacker able to break today's TLS: sessions and content are protected by the post-quantum layers, and a running client authenticates the server with pinned keys. (An attacker who controls TLS while the app is being loaded is a different threat: see malicious client delivery in 13.2.)
- **Harvest now, decrypt later.** Content and session keys are protected by hybrid post-quantum key encapsulation.
- **Operator policy change.** Stored content stays confidential regardless of future policy, because the operator holds no keys.

13.2 What is mitigated, not eliminated

- **Malicious client delivery.** A web application is delivered by its operator, over TLS. An operator — someone compelling it, or anyone who controls TLS at that moment — could serve modified code to a user, including different pinned keys. Each release manifest is signed with the uOS release key, and anyone can check what a host serves against it (Section 14.1); this makes tampering detectable, but a user who does not check is not protected on that visit.
- **Metadata.** The server sees who belongs to which conversation, workspace and meeting, and when activity happens. Sealed sender, padding, coarse timestamps and short-lived presence reduce this; they do not remove it.
- **Key distribution.** Key substitution is detected by the key-transparency log (Section 5.5), except when the server shows different logs to different users whose views never meet, or invents a new account in someone's name; comparing safety numbers covers both. Organisation escrow keys have the same protection and the same limit: an escrow key signed by an invented owner account is detected by the real owners, not prevented (Section 10.4).
- **Offline guessing of secrets.** An attacker with a copy of the database can attempt offline guesses against the wrapped bundle or the login verifier (Section 5.1). Argon2id makes each guess expensive; it cannot save a guessable secret. Accounts that have not logged in since the login gate was introduced still hand out their bundle until their next login.
- **Desktop Agent.** A compromised server can still trigger the agent's read-only tools within the working directory (Section 12). It cannot run commands, change files or open web addresses without the owner's signature.
- **Phishing.** Someone who obtains a user's secrets can sign in as them.

- **AI requests.** The chosen AI provider sees the context of each request, and the relay's no-logging property is a policy of the server (Section 12).
- **Availability.** A compromised or coerced service can refuse to serve data. It cannot read it.

13.3 Out of scope

A compromised user device or browser, physical coercion of the user, and weaknesses in the underlying standards themselves. Symmetric primitives with 256-bit keys are considered secure against quantum attack; if that assessment changes, the suite mechanism (Section 6.2) is the migration path.

14. Transparency and verification

14.1 Verifying the client

Everything that protects the user runs in the browser and can be inspected there. Beyond inspection, each release is built deterministically — repeated builds of the same source produce identical files — and published with a manifest of the SHA-384 hash of every file (`/release-manifest.json`), while the page loads its code with subresource integrity. Every manifest is signed with the uOS release key — a hybrid ML-DSA-87 + Ed25519 key kept off the served infrastructure, whose public half is published with this paper (key id `666MWqG6o0cwguPHVNYgspQDJ9bLtMvUTk/UxeA1YoY=`) and in the repository. A verifier (`scripts/verify-release.mjs`) checks a host's signed manifest against that key and compares the served files byte for byte. The remaining step is publishing each release's manifest hash in an independently operated log, so that serving different releases to different users leaves public evidence.

14.2 The server

The server's source is not public. In this architecture the server is not the user's trust anchor: the guarantees in Section 13.1 hold against a fully malicious server. What the server can accept and send is defined by the published client and protocol. Security researchers, auditors and academics can request scoped access to the server source for review.

14.3 Engineering discipline

Every release runs automated gates before it reaches production: known-answer tests of every primitive across implementations; a coverage check against the target database that fails if any column holding a key or ciphertext is not handled by identity rotation; and the authentication, cryptography, server and HTTP test suites. The production release is refused if any gate fails.

14.4 Deployment options

Organisations choose where uOS runs:

- **The uos.sh platform**, operated by us, where organisations create their own platforms next to personal users.
- **On their own premises**, operated by the organisation on its own infrastructure, with the same code, the same client and the same release manifests — and, where required, the CNSA suite.

The verification chain works the same way in both cases: every release is published with a manifest of its files, so an organisation running uOS itself can check that what it deploys is exactly what was released, and its users can check what their browser receives. In an on-premises deployment the organisation, not us, operates the server; the end-to-end guarantees in Section 13.1 then hold against that operator in the same way.

15. Privacy and data protection

- **Minimal registration.** No phone number, legal name or identity check. An e-mail address is optional; if given, it is stored encrypted under a server key and used only for the security and service messages the user opts into.
- **No tracking.** No cookies for tracking, no advertising identifiers, no third-party analytics, no device fingerprinting, and no external fonts or scripts in the application.
- **Personal data, stated precisely.** Under the GDPR, public keys, connection metadata and short-lived logs can be personal data. uOS minimises them by construction and honours data-subject rights for what it holds.
- **Hosting.** The uos.sh service is hosted by Infomaniak in Switzerland, a country with an adequacy decision of the European Commission under the GDPR. Subprocessors are listed in a published, versioned register: the hosting provider, AI providers only when a user selects them, and push services only when a user opts in. Organisations can instead run uOS on their own premises (Section 14.4). A revision of the Swiss surveillance ordinance (VÜPF), proposed in 2025, would extend identification and data-retention duties to more online services. Under any legal regime, the operator can hand over only what it holds — the metadata in Section 9 — because no identity is collected at sign-up and content is encrypted with keys the operator does not have. We follow the revision and will adapt where the service is hosted if its final form requires it.
- **Legal requests.** Requests are validated for jurisdiction and legal basis before anything is produced, and aggregate numbers are recorded for a public transparency report.

16. Scope and limitations

uOS is in Alpha. The following limitations are known and are being worked on; they are stated here so that no reviewer has to discover them.

- **Older accounts.** A small number of accounts created before the random-seed identity model still use the legacy derivation, in which keys follow directly from the secrets, until their next login, which upgrades them (Section 5.3). Until then, their data remains open to offline guessing of the original secrets. Content they created earlier and that someone copied before the upgrade stays exposed in that copy.
- **Older boards and posts.** Private boards created before field-level encryption stay readable to the server until their owner or an administrator next opens them. They are encrypted automatically at that point, including comments, attachments and activity. Public boards stay readable by design. Early uLine posts are sealed when their author next opens the feed.

- **Meetings.** Calls are peer to peer for up to about six participants and need a current browser that can encrypt media frames (Chrome, Edge, Firefox or Safari). The connectivity (STUN) servers used to establish calls are currently operated by a third party, which sees participants' network addresses.
- **TLS.** uos.sh offers the hybrid TLS group `X25519MLKEM768`, which current Chrome, Edge, Firefox and Safari use. Older browsers fall back to a classical key exchange. Once the app is running, no content and no sensitive request depends on TLS, because the post-quantum layers above protect both. The delivery of the app itself does depend on it (Section 13.2).
- **Client verification.** Release manifests are signed and verifiable, but not yet published in an independent log, and browsers do not check them automatically (Section 14.1).
- **Mobile apps.** The native iOS and Android apps are in development and support the v2 suite; the web application is the supported way to use uOS on phones today.
- **Personal profiles** are readable by the server by product choice, because users publish them to others.
- **Classical primitives.** Sealed-sender delivery tokens use RFC 9474 RSA blind signatures; their blindness holds unconditionally, and a quantum attacker could forge tokens (affecting abuse limits), not unlink them. Web Push uses the classical encryption of the Web Push standard and carries no content.
- **No independent audit yet.** The cryptographic design and implementation have not yet been reviewed by an independent auditor.

17. Roadmap

The next steps, in order of priority:

1. An independent security audit of the cryptographic design and implementation, with the report published.
2. Publication of each release's signed manifest in an independently operated log, and automatic verification by the Desktop Agent or a browser extension.
3. Self-hosted call connectivity servers.
4. Key-transparency gossip with independent witnesses.
5. Alignment of the composite constructions with the IETF standards (a future suite).
6. Native mobile apps on the v3 suite with the login gate, and the CNSA suite for on-premises and dedicated deployments.

18. Business principles

- **Personal use is free**, without advertising, data sale or behavioural profiling — at every stage of the product.
- **Organisations pay per seat**, with non-profit and education terms, on the uos.sh platform or for a deployment on their own premises.
- **Our revenue never depends on reading content.** For encrypted content we cannot read it; the business model and the security model point the same way.
- **During Alpha**, organisations can use uOS free of charge in exchange for structured feedback.

19. Conclusion

uOS starts from a position that most software treats as a trade-off: that a complete, collaborative workspace can be built without its operator being able to read it. Post-quantum cryptography, keys held by users, sealed metadata where possible and a clear account of what remains visible make that position concrete. We invite organisations, security reviewers and early users to test it with us.

Contact: patrick@ultimaos.com · https://ultimaos.com · Sign up and log in at https://uos.sh

APPENDICES

A. Primitives and parameters

Primitive	Standard	Use in uOS
ML-KEM-768 / ML-KEM-1024	FIPS 203	Key encapsulation (v2 / v3), always with X25519
ML-DSA-65 / ML-DSA-87	FIPS 204	Signatures (v2 / v3), always with Ed25519
X25519, Ed25519	RFC 7748, RFC 8032	Classical halves of every hybrid operation
XChaCha20-Poly1305	RFC 8439; draft-irtf-cfrg-xchacha	Authenticated encryption, 256-bit keys, 192-bit random nonces
HKDF-SHA256 / HKDF-SHA384	RFC 5869	Key derivation, domain-separated labels bound to the suite
Argon2id	RFC 9106	Wrapping key from secrets (256 MiB, t = 3, p = 2); PIN hash; recovery-file passphrase
HMAC-SHA256	RFC 2104	Peppered identifier lookup; contact-list lookups
AES-256-GCM	FIPS 197, SP 800-38D	uMeet media frames, keys from ML-KEM-1024 + X25519 per call and direction
SHA-384	FIPS 180-4	Release manifests, subresource integrity
RSA blind signatures	RFC 9474 (RSABSSA-SHA384-PSS-Randomized)	Unlinkable delivery tokens for sealed sender
ML-DSA-87 + Ed25519	FIPS 204, RFC 8032	Release and Desktop Agent update signatures (uOS release key)
Merkle tree (SHA-256)	RFC 6962	Key-transparency log of account keys
Shamir secret sharing	—	Recovery shares; organisation escrow key
TLS 1.2 / 1.3	RFC 5246 / RFC 8446	Transport and app delivery; hybrid `X25519MLKEM768` offered on TLS 1.3

The RSA blind signatures protect the *unlinkability* of delivery tokens, not the confidentiality of any content.

B. Key sizes by suite

Value	v2	v3-hybrid
KEM public key	1,184 B (+ 32 B X25519)	1,600 B, composite
KEM ciphertext per recipient	1,088 B (+ 32 B)	1,600 B, composite

Value	v2	v3-hybrid
Signing public key	1,952 B (+ 32 B Ed25519)	2,624 B, composite
Signature	3,309 B (+ 64 B)	4,691 B, composite

The sizes are the value bodies; every v3 value carries one additional suite-tag byte (for example, a v3-hybrid KEM public key is 1,601 bytes on the wire). v2 values keep their original untagged encoding, so existing data is unchanged.

C. Network surface

Path	Purpose
`/ws`	All user activity: encrypted, signed frames in a post-quantum session
`/api/ws/fallback/*`	The same session over HTTPS long-poll where WebSockets are blocked
`/api/core/hello`, `/api/core/auth_proof`, `/api/core/kdf_params`	Login challenge, proof and salt lookup (lookup sealed with the hybrid KEM)
`/api/core/kdf_unlock`	Release of the wrapped bundle against the login key (sealed)
`/api/core/set-pin`, `/api/core/verify-pin`, `/api/core/change-pin`	PIN management and checks (sealed)
`/api/account/*`	Unlock, change of secrets and identity rotation (sealed), including ticket-gated ciphertext transfer
`/api/uploads/{ticket}`	Bulk upload of ciphertext with a server-signed, single-use ticket
`/api/invites/*`, `/api/platform/*`	Public invitation and platform pages
`/api/profile`, `/api/server/config`	Profile fallback; public server keys and push key
`/api/contact`, `/api/contact/challenge`	Contact form with proof of work
`/api/rss-proxy`	Allow-listed fetch of public news feeds
`/s/*`, `/sdk/*`	Published Studio sites; app and asset bundles
`/healthz`	Liveness

D. Glossary

Core account — a person's cryptographic identity. **Platform** — an isolated workspace with its own members and settings. **Seed bundle** — the random material from which an account's key pairs are derived. **KEK** — key-encryption key, derived from the user's secrets. **CEK** — content-encryption key of a single re-

source. **Suite** — a fixed set of algorithms and parameters. **Epoch** — one generation of a key, replaced on rotation. **Escrow** — an organisation-held recovery key for selected business content. **Sealed sender** — delivery without a stored sender identity. **Forward secrecy** — protection of past messages if a long-term key is later compromised.

uOS

https://ultimaos.com · patrick@ultimaos.com

Sign up and log in at https://uos.sh

UltimaOS