SECURITY · WHITEPAPER
V1.0 · MAY 2026

SECURITY & PRIVACY

We cannot read your data at rest.

How ARBI protects confidential matter material — the customer-held encryption keys, the in-memory-only decryption boundary, and every control around them.

FOR

Security, IT, compliance and procurement teams evaluating ARBI for use with sensitive legal, dispute-resolution and matter material.

Arbitration City Ltd
Juxon House · 100 St. Paul's Churchyard
London EC4M 8BU · United Kingdom
security@arbi.city

FOREWORD

A note from the founder

Why customer-held keys, not vendor promises — and what that costs us to build properly.

I spent twenty years inside law firms before founding ARBI. I have seen the cost of a single misplaced document, an attached file forwarded one too many times, an AI tool that quietly trained on a client's draft. Lawyers don't trust software the way engineers do. They trust other lawyers, and they trust mathematics. So we built ARBI to need as little trust as possible — by making it mathematically uninteresting for us to read your data.

The shorthand we use internally is **"we cannot read your data at rest."** What it means precisely is this: every workspace has its own symmetric encryption key; every member of that workspace holds a wrapped copy of that key, sealed to their personal X25519 public key. When a request arrives, your client sends the key to the server inside another sealed envelope. The server unwraps it in memory, does the work, returns the answer, and lets the key go. We never write it down. We never store it. The bytes on our disk are ciphertext; without a member's key, no key recovery procedure or court order can change that.

This document is the long-form version of how that is implemented, and — just as importantly — where the boundaries of the claim lie. There are things customer-held keys cannot protect you from: a compromise of your own device, an over-eager LLM prompt that quotes a confidential paragraph back at you, a colleague who deliberately copies content into a screenshot. There are also corners of our own stack that are still being hardened. Throughout this whitepaper, you will see `GAP` markers where current code does not yet match the strongest possible posture and `PLAN` markers for items on the roadmap. We have deliberately not airbrushed those.

If the rest of the document leaves you with questions it hasn't answered, please write directly to me at security@arbi.city.

Dmitri Evseev
Founder & CEO, Arbitration City Ltd · London

CONTENTS

What's inside

Nine parts and three appendices. Read straight through, or jump to whichever question you came with.

PART 1

The promise

What "we cannot read your data at rest" actually means — what the claim covers, where its edges are, and the threats we set out to defeat with it.

1.1 What we mean by “at rest”

Precision matters more than slogans. Here is the exact property we set out to guarantee — and three properties we deliberately do not.

Every workspace in ARBI has its own 256-bit symmetric key. That key encrypts every confidential field of every document, message, conversation, annotation and custom-field value belonging to the workspace. The encrypted payloads live in Postgres, MinIO and Qdrant. **Nowhere in our infrastructure is the workspace key stored in plaintext.**

It exists only as SealedBox-encrypted blobs in Postgres — one per workspace member, encrypted to that member's X25519 public key — and, briefly, inside a request-scoped Redis session entry that requires the member's JWT signature to unwrap. "At rest" means the moment no member is actively making a request:

- The bytes on disk are ciphertext.
- The unwrapping material for any one workspace is not stored anywhere we operate.
- No combination of database access, MinIO root credentials, or a court order against Arbitration City Ltd alone can recover plaintext.

To read a workspace's content, an active member must log in, derive their keypair from their master password, and supply an unwrapped form of the workspace key to the server inside a sealed envelope. The server unwraps it in memory for one request, performs the operation, and lets the key go. Plaintext never lands on disk or in a log line.

Three things this property is *not*

Not "we never see plaintext, ever." When you ask ARBI to summarise a document or answer a question, the server has to decrypt the relevant chunks in memory to feed them to the model. For the duration of that request, in process memory, plaintext exists. We do not write it down, log it, or send it to a third party (unless your deployment has been configured to use a third-party model). But pretending the plaintext is never seen would be false.

Not "your device cannot leak." Customer-held keys defeat server-side compromise. They cannot defeat a compromised endpoint, a colleague who screenshots a passage, or a malicious browser extension running with full origin permissions. The mathematical guarantee stops where your device's trust boundary stops.

Not "every byte we operate on is encrypted." Some operational data — your email address, the names of workspaces you belong to, billing records, retention timers, and the *vectors* (raw floats) that document chunks embed to — is stored without workspace-key encryption. We discuss exactly what falls in this category in §2.2, including what each field reveals and why.

> **WHY WE SAY THIS OUT LOUD** Most security marketing collapses these three caveats into "zero-knowledge encryption." We don't, because procurement teams and law-firm CIOs notice — and because customers who eventually find a gap that wasn't disclosed lose trust faster than customers who knew about it from day one.

1.2 Threat model and scope

Concretely: which adversaries are we designing against, which are we not, and what falls to your own security posture rather than ours.

Adversaries we defeat

External attacker against our servers
An attacker who compromises our database, object storage or vector store sees only ciphertext for confidential fields. Without an active member's session, no unwrap is possible.

Malicious or curious insider at ARBI
No one at ARBI can read your data — not even with full production access. That access is gated by mTLS, WireGuard and secret-manager-issued credentials, and even at the database layer the confidential rows are encrypted.

Subpoena or legal compulsion against ARBI alone
We cannot produce plaintext we don't have. Any compelled disclosure would consist of ciphertext, metadata, and the wrapped keys — not the content.

AI provider seeing your prompts
Our default deployment runs every model locally on our London GPUs. The prompt and the document context never leave our network unless the deployment has been configured to use a remote model.

Adversaries that fall outside our boundary

A compromised user device
Malware on a member's laptop, a keylogger, or a hostile browser extension with origin permissions can observe whatever the legitimate user can observe. Defended by your endpoint posture, not by us.

A user voluntarily exporting
Once a member exports a document, copies text into another tool, or shares a screenshot, ARBI's controls no longer apply. We provide audit trails of what was opened but cannot retroactively contain leakage.

Password reuse or weak passwords
Your workspace key is unwrappable by anyone holding your derived encryption private key — which is derived from your master password via Argon2id. If your password is in a breach corpus, an attacker with your public key may be able to brute-force offline.

Quantum-capable adversary
X25519 and XSalsa20-Poly1305 are not post-quantum. We track NIST PQC progress but have not migrated yet — the cost-benefit for the legal-matter time-horizon does not yet justify the engineering complexity.

A two-line summary

If *your* account is safe, ARBI is safe. If *our* servers fall, ARBI is still safe. The dangerous combination is your account *and* our servers both compromised at the same time — and even then, only what your account had access to, never the broader corpus.

PART 2

Architecture

A single request, end to end — and what is in our database that is not encrypted with the workspace key.

2.1 Request dataflow

From your browser to the workspace key to the model and back. Six steps, with the cryptographic boundary visible at each.

1. **You sign in.** Your browser derives your encryption keypair from your master password via Argon2id (libsodium `crypto_pwhash`) and proves possession to the server with an Ed25519 signature over `email|timestamp`. The server checks the signature against your stored public key and mints a short-lived JWT.
2. **The server issues an ephemeral session keypair.** A fresh X25519 keypair is created. The public half travels back to your client embedded in the JWT (as the `sk` claim). The private half is wrapped with the SHA-256 of the JWT signature and stored in Redis — neither half alone is enough to unwrap it.
3. **You open a workspace.** Your browser fetches your wrapped workspace key (one SealedBox per workspace, encrypted to your X25519 public key), unwraps it locally with your encryption private key, and re-seals it to the server's ephemeral session public key.
4. **You make a request.** Each API call carries the re-sealed workspace key. The server validates the JWT, retrieves the wrapped session private key from Redis, unwraps it with the JWT signature hash, then unwraps the workspace key.
5. **The server does the work.** It instantiates a NaCl SecretBox cipher from the workspace key, decrypts only the chunks it needs, runs inference (locally on our GPUs, or via a remote model if the deployment is configured for one), and produces a response. Everything decrypted lives only in process memory.
6. **The response is encrypted on the way back to storage.** The assistant's reply and any new content are encrypted with the same workspace key before being committed to Postgres or MinIO. The session ends; the unwrapped workspace key is dropped from process memory at the end of the request.

THE TWO-FACTOR SESSION Splitting the session secret across the JWT signature (held by the client) and Redis (held by us) means that compromising either Redis alone or the JWT alone yields nothing. An attacker would need to steal both at the same instant *and* the workspace key your client just supplied — and the workspace key changes nothing about another workspace's safety.

2.2 What we hold, what stays opaque

Exact inventory. Anything that is not workspace-key-encrypted is listed here, in full, with what it reveals.

Data	Where it lives	State
Document file bytes	MinIO	WORKSPACE-KEY ENCRYPTED
Document title, author, file name, parsed text	Postgres	WORKSPACE-KEY ENCRYPTED
Chunk text & chunk metadata payload	Qdrant	WORKSPACE-KEY ENCRYPTED
Messages, conversation titles, tool calls, trace steps	Postgres	WORKSPACE-KEY ENCRYPTED
Tag instructions, annotation notes, citation statements	Postgres	WORKSPACE-KEY ENCRYPTED
Direct messages	Postgres	E2E TO RECIPIENT X25519
Wrapped workspace keys	Postgres	SEALEDBOX TO USER X25519
Backups of all of the above	Volume snapshots	INHERITS FIELD-LEVEL ENCRYPTION
Chunk embedding vectors (raw floats)	Qdrant	PLAINTEXT FLOATS · NOT REVERSIBLE
User email, name, public keys	Postgres	PLAINTEXT · IDENTITY
Workspace IDs, membership graph, roles	Postgres	PLAINTEXT · ACCESS CONTROL NEEDS IT
File sizes, timestamps, billing records	Postgres	PLAINTEXT · OPERATIONAL
Auth events, OTEL traces, metrics	Loki / Tempo / Prometheus	PLAINTEXT · CONTENT STRIPPED

On the embedding vectors: arrays of ~1,024 floats per chunk. Not human-readable and not practically invertible to source text. We do not currently encrypt them at rest because doing so would prevent the vector store from serving its function — encrypted-vector-search research is on our backlog. PLAN

PART 3

Encryption

The crypto in detail — primitives, parameters, where each key lives, and what is and isn't covered.

3.1 Per-workspace symmetric keys

One 256-bit symmetric key per workspace. The unit of confidentiality matches the unit of collaboration.

We do not encrypt with one key per document, or one key per user, or one key per matter. We encrypt with one key per workspace. A workspace is the natural unit of confidentiality in a law firm — a matter, a deal, a client engagement — so the cryptographic boundary lines up with the access-control boundary.

CIPHER	KEY SIZE	NONCE	AUTH
XSalsa20-Poly1305	**256 bits**	**192 bits**	**Poly1305 MAC**
NaCl SecretBox	32 random bytes	Random per ciphertext	Tamper-evident

How a key is born

When you create a workspace, your browser generates 32 random bytes using `sodium.randombytes_buf` (libsodium's CSPRNG, which on the web uses `crypto.getRandomValues` under the hood). That is the workspace key. It is immediately wrapped to your X25519 public key with NaCl SealedBox and sent to the server inside the `wrapped_key` field of a workspace-creation request. The raw 32 bytes never travel over the wire and are never persisted by the server.

How a key is shared

When you invite a colleague to the workspace, your browser unwraps the workspace key with your encryption private key, re-wraps it with the colleague's X25519 public key, and uploads only the new wrapped blob. The server stores one wrapped copy per member. To remove a member, we simply delete their wrapped row — their *future* access ends instantly. (Past content they already decrypted is, of course, beyond cryptographic recovery; this is true of every E2E system.)

What the key encrypts

Every confidential field in the workspace flows through one of two helpers — `encrypt_string` for short fields (titles, messages, annotations) and `encrypt_and_save_file` for binary blobs (uploaded documents). Both build a SecretBox cipher from the workspace key and apply a random per-ciphertext nonce. The list of confidential fields is enumerated in our backend's `CONFIDENTIAL_FIELDS` constant and applied uniformly at the database, object-store and telemetry layers.

```
# src/core/encryption.py (paraphrased)
def encrypt_string(plaintext: str, cipher: SecretBox) -> str:
    nonce = random(24)              # 192-bit nonce, fresh per ciphertext
    ct    = cipher.encrypt(plaintext.encode(), nonce)
    return b64encode(ct).decode()
```

3.2 Key wrapping with X25519 SealedBox

How a workspace key gets to each member without us ever seeing it in plaintext.

Every user has two long-term keypairs derived from their master password: an Ed25519 signing keypair (for authenticating to the server) and an X25519 encryption keypair (for receiving workspace keys). Only the public halves leave the device; the private halves are stored encrypted-at-rest in browser IndexedDB under a non-extractable WebCrypto AES-GCM key (§7.2).

SealedBox in three lines

NaCl SealedBox is anonymous public-key encryption built from X25519 ECDH plus XSalsa20-Poly1305:

1. **The sender** generates an ephemeral X25519 keypair, derives a shared secret with the recipient's public key, and encrypts the workspace key under that shared secret.
2. **The wire format** is the ephemeral public key prefixed to the ciphertext. The ephemeral private key is destroyed immediately after encryption.
3. **The recipient** reads the ephemeral public key, derives the same shared secret with their long-term private key, and decrypts.

Property: the sender's identity is unauthenticated (SealedBox is anonymous by design) — which is fine, because the sender of a workspace key invite is identified by an out-of-band channel (their authenticated API call). What matters is that **only the recipient can decrypt**, and that the server, which sees only the ciphertext, cannot.

Where wrapped keys are stored

In Postgres, on the `workspaceusers` table. One row per (user, workspace) pair. Each row carries a `wrapped_key` column containing the SealedBox blob. The same row carries the user's role (owner / collaborator / guest) for access control. Removing a member is a single `DELETE`.

Password change

When you change your master password, the browser derives the new keypair, unwraps each workspace key with the old private key, re-wraps it with the new public key, and uploads the bundle to `POST /v1/user/change_password`. The server replaces `users.encryption_key_pub` and the set of `workspaceusers.wrapped_key` rows in a single transaction. Direct-message ciphertext is rotated through a separate client-driven path on the next session.

> **WHAT WE DON'T CLAIM** If you forget your master password and we have no other recovery proof on file (recovery file, agent recovery key, SSO link), there is nothing we can do. The

3.3 The two-factor session model

A real two-factor split: the JWT signature your browser carries, and a Redis-stored secret only we hold. Either alone is useless.

A typical session token gives an attacker who steals it full impersonation. We split the session secret across two stores: the JWT signature, which lives only in the browser; and a wrapped session private key in Redis, which we hold but cannot unwrap without the matching JWT signature. Stealing one is not enough.

Login

When you sign in, the server creates an **ephemeral X25519 keypair** tied to your session. The public half is embedded in the JWT as the `sk` claim and sent back to your client. The private half is wrapped using the SHA-256 of the JWT's signature and stored in Redis under `session:{deployment}:{sk_pub}` with a TTL of about four hours.

Every subsequent request

1. Your client sends its API call. The Authorization header carries the JWT (whose signature is computed by our server with the deployment's signing key — TPM-backed where available, see §6.4).
2. Our server validates the JWT signature, then hashes that signature with SHA-256 to recover the wrapping key.
3. It looks up Redis for the wrapped session private key under the `sk_pub` from the JWT.
4. It unwraps the session private key in memory.
5. For workspace-scoped calls, the request body also includes the workspace key, sealed to the session's public X25519 — the server unwraps that with the session private key.

Why both factors matter

If the JWT leaks but Redis is intact
The attacker can authenticate API calls — but every workspace-scoped operation requires the session-sealed workspace key, which only the legitimate client has. No data egress.

If Redis is exfiltrated but the JWT isn't
The attacker has wrapped session private keys with no corresponding JWT signature to unwrap them. Garbage.

Session limits
Each user is capped at ten active sessions; the oldest is evicted on each new login. Stale Redis entries are cleaned up at login.

JWT replay window
Tokens expire on a 4-hour wall-clock; refresh re-derives a fresh ephemeral keypair. Each JWT carries a random `jti` so two minted in the same second are still distinct.

3.4 In-memory-only decryption

Where plaintext exists, how long it lives, and the guarantees we can and cannot make about Python memory.

When a workspace key arrives via the session-sealed envelope, the request handler instantiates a `SecretBox` object from it and attaches it to a per-request `WorkspaceContext`. Any decryption that happens during the request — fetching a document's parsed text, decrypting a chunk for the LLM, decrypting a citation's source statement — goes through that one context.

At the end of the request, Python's reference counter releases the context, and the underlying `bytes` object containing the workspace key becomes eligible for garbage collection. There is no on-disk cache, no Redis write-back of the unwrapped key, no log line carrying it.

What we deliberately don't say

We do not claim the key bytes are *zeroised* from memory. Python provides no guarantee of memory wiping for byte strings; `memset`-style overwriting requires C-extension gymnastics and the operating system may have already paged the page to swap by the time we'd try. Instead of pretending we have that property, we make the access window as small as possible:

- The wrapped key is unwrapped per request, not per process.
- Background tasks that survive a request boundary (long-running document parse, batch tag application) receive the workspace key through a deployment-public-key-sealed envelope that they themselves unwrap in their own process, on their own schedule.
- Process memory is never written to disk via core dumps in production (set in our systemd hardening).

Telemetry confidentiality

Equally important: when decrypted content passes through an OTEL-instrumented function call, the OTEL exporter is configured to drop a hard-coded list of confidential field names from every span attribute and log record. That list is defined once in `src/constants.py` and applied uniformly by a `separate_confidential_data` hook in our telemetry setup. New code paths that handle confidential content must add the relevant field name to that list — there is no opt-in to logging plaintext.

```
# src/constants.py (excerpt)
CONFIDENTIAL_FIELDS = frozenset({
    "query", "user_query", "response", "history",
    "content", "title", "file_name", "note",
    "statement", "doc_title", "doc_author", "system_instruction",
    "tool_calls", "chunks_to_keep", "focus", "learnings", ...
})
```

3.5 Public workspaces — honest disclosure

One pattern in our codebase that does not meet the "we cannot read at rest" bar. What it is, why it exists, and what we are doing about it.

GAP Workspaces explicitly created as *public* store their symmetric key as plaintext base64 in `workspaces.shared_key`. Documents in a public workspace are encrypted at rest with that key, but the key itself is recoverable by anyone with raw Postgres access. Public workspaces are opt-in and never the default.

What public workspaces are for

A small number of customers — typically large legal-community deployments — want a shared "common" workspace that any signed-in user of their deployment can read without an explicit invitation. The use case is precedent libraries, firm-wide know-how, shared templates. The trade-off is membership becomes implicit (anyone with deployment access), so per-user key wrapping doesn't make sense.

What it should be

The plan, and the original design intent visible in the in-code docstring, is to wrap the public workspace's symmetric key with the deployment's X25519 public key. Then the deployment private key — which lives in our TPM or in an out-of-tree secret — would be needed to unwrap, restoring the "Postgres alone is not enough" property even for public workspaces. PLAN

What this means for you

- If you do not create public workspaces, this gap does not affect you. The default in the UI is **private**; you must explicitly opt a workspace into *public* via an admin-only flow.
- If you do use public workspaces, treat their contents as you would treat content stored in a regular per-tenant database with Postgres-only encryption at rest. That is still better than most SaaS — but it is materially weaker than our default model.
- Direct messages, private workspaces, and private content within a public workspace remain fully wrapped to user X25519 keys. The gap is specifically the public workspace's *shared* symmetric key.

WHY THIS SECTION EXISTS Our existing internal compliance documentation describes public workspaces as wrapped with the deployment key. The code does not implement that yet. We chose to publish this whitepaper with the gap visible rather than ship the marketing claim ahead of the code. Tracking issue is open with security@arbi.city.

3.6 Data in transit

TLS at the edge, mTLS in the middle, and the bits where we are still pinning the floor.

External traffic

All ARBI Docs traffic terminates at Cloudflare's edge and is re-terminated at our origin nginx by certificates issued via Let's Encrypt DNS-01. The pairing is configured as full-strict mode at Cloudflare. Cloudflare's tenant-level defaults negotiate TLS 1.2+ with TLS 1.3 preferred; we have not yet pinned this in our origin nginx configuration — that is on the hardening backlog. GAP

Customer-facing mTLS for ARBI Box

Self-contained appliance deployments use mTLS between the customer's network and our admin plane. Our origin nginx-proxy issues client-cert-required policies on the relevant subdomains, and the client certificate chain lives on the customer's appliance with read-only Docker secrets at file mode 0400.

Internal service-to-service

App ↔ central-services traffic (LLM proxy, embedder, reranker, parser) uses mTLS with client certificates issued by our internal CA. Where mTLS is not in play, traffic stays on private Docker overlay networks on dedicated bare metal — but we have not enabled Docker Swarm's overlay-network encryption (`--opt encrypted`) on those networks. GAP

Database connections

The Ansible-managed PostgreSQL cluster requires `hostssl ... scram-sha-256 clientcert=verify-ca` in `pg_hba.conf`. Single-node swarm-managed Postgres allows non-SSL connections from inside the same overlay; this is fine for the local-only single-host case but is something we are aligning with the cluster posture.

Administrative access

Our admin subdomains (Portainer, Grafana, OTEL, registry, HAProxy stats) are unreachable without WireGuard *and* a client certificate. Cert provisioning is via a self-service API on our central server, audited.

> **WHAT WE'RE MOVING ON NEXT** The three GAPs flagged above — pinning TLS minimum at origin nginx, adding HSTS on our edge, and enabling overlay-network encryption — are scheduled for the Q3-2026 hardening pass. The functional security is provided today by Cloudflare's defaults and the private-network topology; the GAP markers flag where our own configuration is not yet authoritative.

PART 4

Identity & access

From a password in a browser to a row in Postgres — how we prove who you are, derive your keys, and enforce what you can see.

4.1 Password-derived keypair

From a master password to an Ed25519 signing key and an X25519 encryption key — the maths, the parameters, and the trade-offs.

Your master password is the root of trust. From it, your browser derives a 32-byte seed using Argon2id, then a deterministic Ed25519 keypair, then an X25519 keypair via standard conversion. The server only ever sees the two public halves.

KDF	SALT	PARAMETERS	OUTPUT
Argon2id libsodium `crypto_pwhash`	**Deterministic** SHA-256(email|domain) · 16 B	**2 ops · 64 MiB** libsodium INTERACTIVE	**32-byte seed** Ed25519 + X25519 deriv.

Why deterministic salt

A random salt per password is the textbook recommendation, because it forces an attacker to compute one Argon2 chain per target. We use a deterministic salt — SHA-256 of `username|deployment_domain`, first 16 bytes — for one specific reason: it lets you re-derive your keypair on any device with just your password and your email. You don't have to back up a per-user salt, you don't have to type a 32-character recovery code on a phone keyboard, and you can recover your account from a clean install on a new laptop.

The cost of that choice is the absence of per-user salt entropy. An attacker who steals your Ed25519 *public* key can build an offline Argon2 dictionary attack against your specific user. So the parameters matter: Argon2id at libsodium's INTERACTIVE preset (2 operations, 64 MiB of memory) makes each guess take roughly half a second on a modern CPU and prevents trivial GPU acceleration. Combined with a password-strength meter that requires entropy in line with NIST 800-63B AAL2 minimums, this is acceptable for our threat model — but customers operating under higher threat (state-level adversaries, particularly sensitive matters) should consider the SSO-only login flow, where the password never leaves the IdP and Argon2 is bypassed entirely.

From seed to two keypairs

```
// Browser-side, packages/arbi-client/src/crypto/sodium.ts
const seed = sodium.crypto_pwhash(
  32, password, salt,
  OPSLIMIT_INTERACTIVE, MEMLIMIT_INTERACTIVE,
  ALG_ARGON2ID13
)
const signingKeyPair    = sodium.crypto_sign_seed_keypair(seed)
const encryptionPubKey  =
sodium.crypto_sign_ed25519_pk_to_curve25519(signingKeyPair.publicKey)
const encryptionPrivKey =
sodium.crypto_sign_ed25519_sk_to_curve25519(signingKeyPair.privateKey)
```

4.2 Login challenge & sessions

Proof of password possession via Ed25519 signature — no hash sent over the wire, no replay window beyond a few seconds.

Local-account login

To log in to a local (non-SSO) account, your browser derives your Ed25519 keypair from your master password (§4.1), signs the string `{email}|{unix_timestamp}` with the private key, and posts the signature with the public key to `POST /v1/user/login`. The server looks up the user by email, checks that the public key matches the one on record, verifies the signature, and confirms the timestamp is within a one-minute skew window.

That is the entire authentication. No password hash crosses the wire. No password is stored server-side. We do not have anything to leak.

> **WHAT YOU CAN'T DO WITH THIS DESIGN** Because the server has no password material, "I forgot my password" cannot be a server-side reset. Recovery uses one of three paths: a recovery file you downloaded at signup (URL-fragment-encoded so the server never sees the raw key); a recovery key wrapped to your Auth0 SSO sub (so an SSO re-link can restore access); or, for organisations, an admin-issued reinvite that creates a fresh account and re-shares workspaces explicitly.

Session establishment

On successful login the server creates an ephemeral X25519 session keypair (the construction in §3.3), embeds the public half in the JWT as the `sk` claim, wraps the private half with the JWT signature hash, and stores the wrapped private half in Redis. The JWT comes back to your browser; the wrapped private half stays in Redis with a 4-hour TTL.

JWT signing

Our JWTs are RS256-signed. On single-host on-prem deployments the signing key can be held in a TPM (`USE_TPM=true`) so the key bytes never appear in process memory. On multi-replica swarm deployments the signing key is a Docker secret at file mode 0400, owned by a non-root user, with a Redis-backed JWKS so any replica can verify any other's tokens.

Token defences

- **Random `jti`.** Each token carries a 64-bit hex identifier so two minted in the same second remain distinct and individually revocable.
- **Agent-token allow-list.** Sensitive endpoints (password change, billing, sharing) refuse agent-issued JWTs. Agents only act on the surface their human owner permits.
- **Email enumeration protection.** The verification endpoint silently succeeds for already-registered emails — attackers can't enumerate accounts via that path.
- **Concurrent-login eviction.** Each user is capped at ten active sessions; re-login from an

4.3 SSO, MFA & permissions

Where Auth0 fits in, what MFA does for you, and the role model that decides who can do what.

SSO via Auth0

ARBI Docs (the SaaS tier) integrates with Auth0 as the primary identity provider. Sign-in flows are OIDC: redirect to Auth0, return with an ID token, exchange for a session. ARBI never sees the user's identity-provider password; Auth0 handles all enrolment, password rotation and federated-IdP integrations on its side. The Auth0 SDK in our React client uses `useRefreshTokens: true` and persists tokens to `localStorage` for cross-tab availability.

MFA

Multi-factor authentication is configured at the Auth0 tenant level — typically enrolling TOTP, FIDO2/WebAuthn or push-app factors. From our application's perspective, MFA is enforced by Auth0 before the ID token is ever minted, so an ID token returning from Auth0 implies MFA was satisfied where required. Our application code itself does not implement a separate second-factor challenge for local Ed25519 logins; the Ed25519 signature is treated as the proof-of-possession factor for those accounts.

> **IF MFA MATTERS TO YOU** Configure Auth0 with MFA required for your tenant, and require all members to sign in via SSO. Local Ed25519 accounts are appropriate for single-user scenarios but do not have a per-login second factor today.

Permissions inside a workspace

Three roles, one per (user, workspace) row in `workspaceusers`:

Role	What they can do
Owner	Manage members and their roles. Edit settings. Delete the workspace (only when no other members remain). All collaborator and guest permissions. Cannot delete content created by others *private* to them, because they cannot decrypt it.
Collaborator	Upload documents, create conversations and annotations, share their own content, apply tags. Can edit shared resources others have created. Cannot manage members.
Guest	Read-only access to shared content. Cannot upload, create or edit. The cleanest pattern for inviting external counsel into a workspace.

Direct messages sit outside the workspace model

One-to-one direct messages between two users are encrypted with NaCl box (X25519 ECDH + XSalsa20-Poly1305) using the recipient's long-term X25519 public key. They are independent of any workspace membership — you can send a DM to any other registered user — and the server

4.4 PostgreSQL row-level security

Defence in depth: even if an application bug let an attacker run a SQL query, the database itself refuses to return rows from another tenant.

Encryption protects content. RLS protects metadata. Every workspace-scoped table in Postgres has explicit policies that filter rows by the requesting user's identity — enforced by Postgres itself.

How it works

1. The app runs as an unprivileged Postgres role (`app_user`) with *no* `BYPASSRLS` privilege. Privileged operations use a separate elevated role on a separate connection pool.
2. Every authenticated request sets a per-transaction GUC: `SET LOCAL myapp.current_user_id = {user_external_id}` .
3. Every policy refers to that GUC via `SECURITY DEFINER` helpers — `is_workspace_member(...)` , `is_visible(...)` , `same_owner_group(...)` .
4. A query that tries to read a row the user has no relationship to silently returns zero rows. No "permission denied" leak.

Tables covered

`users` , `workspaces` , `workspaceusers` , `docs` , `tags` , `doctags` , `convos` , `messages` , `notifications` , `configurations` , `contacts` , `doc_similarities` , `projects` . Policies are reapplied idempotently on every boot.

Background tasks & agents

Jobs that operate outside an HTTP request open their database session inside `rls_session_for_background_task()` , which sets the same GUC to the originating user's ID. Persistent autonomous agents are ordinary `users` rows with a `parent_user_id` ; they get their own `workspaceusers` row in each workspace and run queries under the same GUC mechanism. There is no "system bypass" mode in the application code.

> **BELT-AND-BRACES** Customer content is already workspace-key-encrypted. RLS adds a second layer: even the *existence* of another workspace's documents is hidden from a cross-tenant query.

PART 5

AI security

What happens when your prompt meets the model — where the model runs, who sees the prompt, and what is and isn't logged.

5.1 Where models run

By default, on our GPUs in London. Optionally, on a third-party endpoint your deployment is configured to use.

Our cluster, our hardware

ARBI's standard tier (and the default for every workspace) runs every model on our own GPU cluster in London. Inference happens on the same trust boundary as our storage layer: the workspace key is unwrapped in process memory, the document chunks are decrypted, the prompt is constructed, the model forward-passes, and the response is encrypted again before it lands in Postgres. There is no external API call, no third-party endpoint, no data exit.

The model catalogue includes general-purpose chat models (gpt-oss family, Q3-VL) and specialist legal models. Each deployment can pin model versions; updates are operator-initiated, never silent.

Optional third-party models

For some matters our customers want the latest frontier model — typically Anthropic Claude or OpenAI GPT — for a specific task. ARBI can route to such providers, but this is a deployment-level configuration, not an automatic default:

- **Local by default.** A deployment's standard backend runs ARBI-hosted models on our own GPUs. No external provider is contacted unless one is explicitly configured in the deployment's model routing.
- **Configured by the operator.** Which models — local or third-party — appear in the catalogue is set in the deployment's model-routing configuration (an internal LiteLLM proxy). On ARBI Box and Flex, that operator is you.
- **Labelled by provider.** Each model is returned to the client with its provider and type, so a user can see whether a model runs locally or on a named external provider before selecting it.
- **Contractually GDPR-bound.** Every third-party provider we make available is under a DPA that prohibits training on our customers' data and limits retention to operational windows the provider documents publicly.

What leaves our network — concretely

If your deployment is configured with a third-party model and a user selects it for a conversation, the system prompt, the user query, and the retrieved document chunks travel to the provider's endpoint over TLS. The decryption happens inside our process (the provider receives plaintext over its own TLS channel); we do not forward ciphertext to them because they have no key. The provider's reply travels back over TLS, and we re-encrypt it under the workspace key before storage.

If no third-party model is configured for your deployment, nothing of yours leaves our network.

WHY THIS IS A WORKSPACE-LEVEL DECISION Different matters have different sensitivity profiles. A precedent-research workspace might happily talk to Claude. A live arbitration

5.2 No training, no content logging

Two related but distinct commitments. Both have specific code paths backing them.

Customer content is not used for training

The models we serve on our own GPUs are either checkpoints we license from upstream providers (with retraining rights staying with the upstream) or open-weights models we run unmodified. We do not collect customer prompts or responses into any fine-tuning pipeline. We have no internal dataset that mixes customer-supplied text with model training corpora. If we ever wanted to train on our customers' data — and we do not — we would need to ask first and provide a clear opt-in.

For third-party models: every provider we integrate is under a DPA that prohibits training on our API traffic. OpenAI's API tier, Anthropic's API tier, and similar all have these terms standard. We surface only providers offering this commitment.

Content is not in our logs

This is the part backed by code. Every function in our codebase that handles confidential payload is wrapped (decorated) with a list of field names to strip from telemetry. The decorators feed into a single `separate_confidential_data()` hook that splits each function's input/output dictionary into a "confidential" half (goes to encrypted DB, never to telemetry) and an "operational" half (timing, status, resource IDs — goes to OTEL).

The list of stripped fields is centralised in `CONFIDENTIAL_FIELDS` (see §3.4) and audited on every PR. Adding a new code path that touches plaintext requires either adding the field to the list or making the function not telemetry-exported.

What our telemetry actually contains

Telemetry signal	Examples of fields included
Authentication events	Login success/failure, IP, user agent, MFA satisfaction, token `jti`. *No password material.*
Admin actions	Member added/removed, role changed, workspace deleted, password rotated. *No keys, no wrapped material.*
Operation timing	API endpoint duration, status code, request size, response size, token counts.
Inference metrics	Model name, tokens-in, tokens-out, time-to-first-token, total latency. *No prompt or response text.*
Errors	Exception class, file:line, anonymised request ID. *Stack traces redact any local variables flagged confidential.*

PART 6

Infrastructure

Where the bits actually live — the London datacentre, the network perimeter, the container posture, secrets, and backups.

6.1 London datacentre

Our own hardware, our own racks, no US cloud provider in the customer-data path.

Unlike most AI service providers, we build and operate our own server infrastructure in central London. Customer data does not transit a US hyperscaler. The bare-metal hosts are pinned by Docker Swarm labels so stateful services land on known nodes; ephemeral application replicas can rebalance freely on top.

LOCATION	TIER	COMPUTE	OPERATOR
Central London United Kingdom	**III-equivalent** Power, cooling, multi-carrier	**NVIDIA GPU** Purpose-built AI cluster	**Arbitration City Ltd** Direct lease, no resellers

Datacentre certifications

The London facility is operated to ISO 27001 (information security), ISO 9001 (quality), ISO 14001 (environmental), ISO 45001 (health & safety) and PCI-DSS. These certifications attach to the datacentre operator and its physical controls — perimeter security, power, climate, access logs, fire suppression. To be clear about scope, they do not extend to the ARBI application platform: ARBI itself does not currently hold a formal information-security certification such as ISO 27001 or SOC 2. We have put our engineering effort into the customer-held-key architecture described in this paper, and we will tell procurement teams plainly where that trade-off matters for them.

Hardware roles

Class	What runs there
arbi-rack-*	Standard application replicas, FastAPI workers, Redis, parser workers.
arbi-whitebox-101/112	Primary & replica PostgreSQL nodes (Patroni VRRP cluster, Ansible-managed).
arbi-whitebox-204/205	MinIO object-storage nodes (erasure-coded across both).
arbi-gpu-*	NVIDIA-equipped inference nodes for our hosted LLM, embedder and reranker.
arbi-edge-*	nginx-proxy origin nodes behind Cloudflare.

Off-prem footprint (audit trail only)

The only off-premise dependency in the customer-data path is for Semaphore (our CI/CD orchestrator) backups, which go to OVH Cloud's UK S3 region (`s3.uk.io.cloud.ovh.net`). That bucket contains CI job history and metadata — no customer content. OVH is an EU-based provider; we have no US-cloud dependency for any customer-data system.

6.2 Edge, TLS & mTLS

How traffic enters the perimeter, gets re-encrypted at the origin, and authenticates itself layer by layer.

Cloudflare edge

Public traffic to `*.arbidocs.com` and `*.arbibox.com` is proxied through Cloudflare in "full strict" mode. Cloudflare provides DDoS mitigation, edge TLS termination, and account-level WAF rules. Our DNS records are managed via the Cloudflare API (provisioned through our own setup-dns script) so changes are auditable through Cloudflare audit logs.

Origin termination

After Cloudflare, traffic terminates again at our origin nginx-proxy via Let's Encrypt certificates renewed by an `acme-companion` container using DNS-01 challenges. Internal traffic between our edge and the application cluster stays on private overlays.

mTLS layers

Origin → admin plane

Subdomains for Portainer, Grafana, OTEL, Docker registry and HAProxy stats require a client certificate. No client cert, no response. The cert chain is provisioned per-engineer via a self-service API on our central server.

App ↔ central API

Per-deployment ARBI applications carry their own mTLS client cert in a Docker secret (mode 0400, non-root) and talk to the central API only over a mutually-authenticated TLS channel.

App ↔ internal services

Outgoing calls to LiteLLM, embedder, reranker and parser services use mTLS with certs from the same internal CA. Cert rotation is automated.

Stream-layer SNI allow-list

The nginx-proxy stream module drops every TLS connection whose SNI is empty or does not match an allow-listed hostname. IP scans, default-DNS misconfigs, and SNI-less probes are blackholed.

PostgreSQL connections

The Ansible-managed Postgres cluster requires `hostssl ... clientcert=verify-ca` in `pg_hba.conf`: server SSL is on, the client must present a CA-issued cert, and password is SCRAM-SHA-256. Single-host swarm-managed Postgres allows plaintext connections within the same overlay network — this is acceptable for the local-only deployment topology but we are aligning the postures. GAP

6.3 Containerisation & isolation

Production containers ship with all capabilities dropped, a read-only root filesystem, and a non-root user. The dev profile relaxes this — production does not.

Standard production posture

Every long-running service in our production swarm stack is composed against a shared YAML anchor that applies the following:

- `cap_drop: [ALL]` — every Linux capability is dropped at container start. Anything that needs a capability has to declare it explicitly with `cap_add`; nothing currently does.
- `read_only: true` on the root filesystem. Writable areas are explicit tmpfs mounts at `/tmp` only.
- `security_opt: ["no-new-privileges:true"]` — set-uid binaries cannot gain privileges.
- Non-root container user. PostgreSQL pins `user: "70:70"`; application containers run as a dedicated non-root user provisioned in the image.

Network isolation

Services join Docker Swarm overlay networks that segregate trust zones: `arbi-net` for application traffic, `central-net` for app↔central, `dc-net` for storage, `office-net` for our office VPN entrypoints. Only the edge tier has public bindings; the rest reach the outside world only through the egress proxy.

OVERLAY ENCRYPTION We have *not* currently set `--opt encrypted` on the data-plane overlay networks. Inter-service traffic to Qdrant, MinIO and Redis on the overlay is plaintext at the L4 layer (the application layer is still encrypted by HTTPS where applicable). Within a single physical datacentre on dedicated bare metal this is a defence-in-depth concern, not a primary protection. `GAP`

What stays as root

A small set of OpenClaw and parser-sandbox containers run as root — they need filesystem-level access to user-provided files during parsing. They have full `cap_drop` but the user namespace remains root. We treat the parser sandbox as a potentially-hostile environment and ensure no key material is reachable inside it (the workspace key is supplied SealedBox-wrapped to the parser's own pubkey, and unwrapped inside an isolated process).

Dev vs prod

Local development containers explicitly opt out of these hardening settings — seccomp and apparmor are unconfined, root filesystems are writable, source code is bind-mounted. Production deployments use the swarm stack file, where all of the above are non-negotiable.

6.4 Secrets & TPM

No secret in git, no secret in source. Two layers: Bitwarden Secrets Manager for runtime, Ansible Vault for at-rest IaC, TPM-backed signing where the hardware supports it.

Runtime secrets via Bitwarden Secrets Manager

Every Docker Swarm stack reads its secrets at deploy time from Bitwarden Secrets Manager (BWS). Each running container has access only to the secrets it needs, mounted as Docker secrets at file mode 0400. There is no `.env` file checked into source. The BWS access token itself is provisioned via an Ansible playbook from a per-host one-time-use token, then rotated.

IaC secrets via Ansible Vault

Anything that must live alongside infrastructure code (DB bootstrap passwords, root CA private keys, internal API tokens) lives in `group_vars/all/vault.yml` encrypted with Ansible Vault AES-256. The Vault password is itself held in BWS. So to read an IaC secret at rest you need BWS access; to use one at runtime you need BWS access. The two-layer arrangement keeps the source repository safely public-cloneable for engineering purposes.

TPM-backed JWT signing

On single-host on-prem deployments (notably ARBI Box) where a TPM 2.0 chip is available, we configure JWT signing to use a TPM-resident RSA key (`USE_TPM=true` , `TPM_KEY_HANDLE=...`). The signing key never appears in process memory — signing operations are dispatched to the TPM and only the signature comes back. If the appliance is stolen, the chip is bound to the motherboard; if the disk is cloned, the TPM key handle is meaningless on different hardware.

For multi-replica swarm deployments where the JWT must be verifiable across replicas (and replicas can't share a TPM), we fall back to a Docker-secret-mounted RSA private key with a Redis-backed JWKS distribution.

Secrets that aren't ours

Customer-supplied credentials (Stripe customer IDs, Auth0 connections, third-party-model API keys) are themselves stored in BWS, in a namespace gated to the customer's tenancy. We treat them as we treat our own production secrets — same mode, same audit trail.

> **WHAT YOU SHOULD EXPECT TO SEE IN AN AUDIT** Ask us for the BWS audit log around the time of any production change; the secret-fetch events are individually attributable. Ask for the Ansible Vault password rotation schedule. Ask for the TPM PCR policy on Box-tier appliances. The answers exist.

6.5 Backups & business continuity

Daily encrypted snapshots, off-site CI metadata, tested restores — and a frank list of what is not backed up.

What is backed up

System	Method & cadence
PostgreSQL (app data)	Daily volume snapshots via hard-link snapshot. 14-day rolling retention on the host filesystem. Field-level encryption inherits to the snapshot.
MinIO (documents)	Daily volume snapshots, same mechanism. Bytes are workspace-key-encrypted before being written, so the snapshot is ciphertext at rest.
Qdrant (vectors + payloads)	Daily volume snapshots, same mechanism. Payload content is encrypted; vectors are not.
Semaphore (CI metadata)	Daily SQL dump replicated off-site to OVH UK S3. 30-day retention. CI metadata only — no customer content.
Configuration / IaC	The Ansible repo itself, Vault-encrypted, is the source of truth and is mirrored to a private GitHub remote.

Restore testing

Restore is exercised on a deliberate cadence: we restore a snapshot to a parallel staging environment, validate the application against it, then tear it down. This catches "the backups work but the restore tooling has drifted" failure modes early.

What is *not* currently backed up

Honesty matters here:

- **The PostgreSQL host VMs themselves** (i.e. the OS layer, not the database) are not backed up. They are rebuilt from Ansible.
- **The nginx-proxy host VMs** are not backed up; they are similarly rebuilt from Ansible and Let's Encrypt re-issues certs on first boot.
- **Edge metrics dashboards** are not retention-prioritised; the time series in Prometheus has its own 30-day window and is not archived long-term.

The implicit promise is "the data layer is durable and recoverable; the compute layer is reproducible and disposable." We think this is the right cost/benefit, but if your compliance regime requires every host to have a backup line in a register, that gap is something to discuss before contracting.

PART 7

AppSec & client

HTTP security headers, browser storage, input validation — and an explicit list of the client-side gaps still open in our hardening backlog.

7.1 HTTP security headers

A strict CSP, no inline scripts, no clickjacking, no MIME-sniffing. What our FastAPI middleware sets on every response.

Every response from the ARBI API passes through a security-headers middleware that applies the headers below. These are not Cloudflare defaults — they are explicit code in our backend that ships with every deployment, including air-gapped Box installs.

Header	Value
Content-Security-Policy	`default-src 'self'` ; `frame-ancestors 'none'` ; `upgrade-insecure-requests` ; no `unsafe-eval` ; no inline scripts
X-Content-Type-Options	`nosniff`
X-Frame-Options	`DENY`
X-XSS-Protection	`1; mode=block`
Referrer-Policy	`strict-origin-when-cross-origin`
Permissions-Policy	`geolocation=()` , `microphone=()` , `camera=()`

Strict-Transport-Security

HSTS is not currently set by our application middleware. In the standard SaaS deployment the Cloudflare edge layer can be configured to inject HSTS, but we have not pinned that in our own configuration; on-prem deployments without a Cloudflare-equivalent layer therefore do not enforce HSTS at the perimeter. We plan to add it to the application middleware. GAP

CORS

The default CORS policy on our SaaS edge is currently permissive (`allow_origin_regex=".*"` with credentials). This was a development convenience that has not been tightened for the public-facing tier. We are in the process of replacing it with an explicit allow-list of `*.arbidocs.com` , customer-tenant subdomains, and integration partners. GAP

Input validation

Every request body in the FastAPI app is validated by a Pydantic `StrictBaseModel` with `extra="forbid"` . Unknown JSON keys are rejected outright. Mass-assignment and parameter-pollution attacks against our request handlers have a tiny surface. XML inputs (for some legacy document formats) go through `defusedxml` , which is hardened against billion-laughs and external-entity attacks.

7.2 Browser storage

Every byte we keep on your device, where it lives, and what guards it.

The honest inventory

What	Where & protection
Ed25519 signing private key	IndexedDB `arbi-crypto/session`. AES-256-GCM-wrapped (random 96-bit IV) under a **non-extractable** WebCrypto AES key. Browser-enforced — even an XSS-level attacker cannot exfiltrate the raw bytes.
Wrapping AES CryptoKey	IndexedDB `arbi-crypto/wrapping-key` as an opaque `CryptoKey` handle. Raw bytes unreachable.
UI state (Zustand)	IndexedDB `arbi-ui-storage`. Workspace and conversation IDs, panel layout, recent doc IDs. *No tokens.*
Auth0 ID/access/refresh tokens	`localStorage`, plaintext. XSS-reachable. Configured this way for cross-tab refresh; we are tracking a refactor to memory-only. GAP
ARBI API access token	`sessionStorage` per tab, plaintext. Cleared on tab close. XSS-reachable.
Selected workspace / pending recovery state	`sessionStorage`, identifiers only.
Workspace symmetric keys	Process memory only. Never persisted to IndexedDB, localStorage, or any other store.

The wrapping construction

Your Ed25519 signing private key is what protects access to all your wrapped workspace keys. So we go to some length to make it hard to steal from a stolen browser profile. The wrapping AES key is generated with the WebCrypto extractable bit set to `false`. The browser refuses to give us — or anyone — the raw bytes. To use the key, the browser performs `decrypt()` via SubtleCrypto with our app's same-origin code in the call stack.

Threat model honestly: passive access to your IndexedDB (someone reads your laptop's disk) gets encrypted bytes plus an unusable handle. Active access (XSS executing in our origin) can *use* the key to decrypt, but cannot copy it out — which means the attack is per-session rather than "exfiltrate once, attack forever."

DM crypto in a Web Worker

Direct-message encryption and decryption run inside a dedicated Web Worker (with Comlink for ergonomics). Two reasons: it keeps message decryption off the main thread so the UI stays responsive, and the worker context is a separate JS realm — even if a UI-thread XSS somehow

7.3 Known gaps & roadmap

A consolidated list of every `GAP` and `PLAN` tagged earlier in this whitepaper — and where each one is in our queue.

Item	Status & planned remediation
Public-workspace key wrapped to deployment X25519 (§3.5)	`GAP` Today stored as plaintext base64. Replacing with deployment-public-key SealedBox; tracked.
Origin nginx TLS minimum version pinning (§3.6)	`GAP` Relies on Cloudflare defaults. Pinning `ssl_protocols TLSv1.2 TLSv1.3` and modern cipher suites at origin nginx in Q3-2026 hardening pass.
Strict-Transport-Security header (§7.1)	`GAP` Adding to FastAPI security middleware with 1-year max-age and includeSubDomains.
Docker overlay network encryption (§6.3)	`GAP` `--opt encrypted` not currently set. Planned alongside next swarm-stack migration.
CORS allow-list tightening (§7.1)	`GAP` Current policy is permissive; explicit allow-list scheduled.
Single-host swarm Postgres TLS (§6.2)	`GAP` Cluster requires TLS; single-host swarm currently allows plaintext on the same overlay. Aligning postures.
Auth0 tokens off `localStorage` (§7.2)	`GAP` Investigating in-memory + refresh-token flow that survives tab navigation without persistence.
Markdown rendering with raw HTML enabled	`GAP` `rehype-raw` enabled without `rehype-sanitize` in AI-output renderer. Adding sanitization with allow-listed tags.
Encrypted vectors in Qdrant (§2.2)	`PLAN` Tracking research into homomorphic / FHE-style encrypted vector search.
Argon2 parameters upgrade (§4.1)	`PLAN` Migration path from INTERACTIVE to MODERATE preset for new accounts; existing accounts re-derive on next password change.

If a gap is a blocker for your procurement, raise it — we will tell you honestly whether and when we plan to close it.

PART 8

Deployment & compliance

Three tiers for three threat appetites — and the GDPR, DPA and data-residency posture that sits over all of them.

8.1 Docs, Box, Flex

Pick the tier that matches who you want holding the keys and the iron.

Tier	What & why
ARBI Docs SaaS · default	**Multi-tenant on our hardware in London.** Same encryption model as every tier — workspace keys held by you, server stores ciphertext. Best for individuals, small teams and matters where our standard security model is sufficient. RTO 24h, RPO 4h.
ARBI Box Dedicated	**Self-contained appliance.** Custom-built hardware, NVIDIA GPU, full ARBI stack including local LLMs. Deploys on-premise at your office or hosted in our London DC with exclusive VPN access to your team. Your IT controls backups, updates, and the TPM-pinned signing key. Can run fully air-gapped where your network policy requires it.
ARBI Flex Enterprise · coming soon	**Runs in your private cloud.** The ARBI stack as Helm charts / Terraform modules, deployed inside your VPC. ARBI CITY has no operational access to your environment. You bring the secrets, the IdP, the storage, and the network policy. Suited to large firms with existing security infrastructure they want to keep.

Same crypto, different operators

Every tier uses the same per-workspace symmetric key + X25519 SealedBox wrapping + in-memory-only decryption model described in Part 3. The difference is who holds the supporting infrastructure:

Storage operator
Docs: us. Box: us or you. Flex: you. The bytes on disk are ciphertext in all cases.

Identity provider
Docs: Auth0 (our tenant). Box: your IdP, federated. Flex: your IdP, direct.

Compute trust boundary
Docs: shared multi-tenant. Box: your hardware. Flex: your cloud account.

Models
All tiers default to local models on the included GPUs. Third-party models, where used, are enabled in the deployment's model-routing configuration.

Support access model

For Docs, we have *no standing access* to customer content. Because every confidential field is encrypted under a workspace key that only your client supplies at request time, we cannot decrypt your documents, messages or conversations from the database or object store — there is no operator path to read them. For Box, we have no access at all unless you initiate a remote-assistance session. For Flex, we have no access full stop — debugging is via support packages you generate and send us.

8.2 GDPR, residency & subprocessors

The legal scaffolding around the technical claims.

Role under GDPR

Arbitration City Ltd is a **processor** under UK GDPR / EU GDPR for the customer content stored in ARBI Docs. Your firm is the controller. Our standard Data Processing Addendum (DPA) is available on request; we can also execute customer DPAs subject to legal review.

Data residency

- **Primary services:** all customer content is processed and stored in the United Kingdom (our London datacentre). No transfer outside the UK for primary services.
- **Third-party AI models:** if your deployment is configured to use a remote model and a user selects it, content traverses that provider's infrastructure (often the United States); provider-specific transfer mechanisms (SCCs, UK Addendum, EU-US Data Privacy Framework) apply.
- **Backups:** all customer content backups are local to our London datacentre. The only off-site replication is CI-job metadata to OVH UK S3.
- **Box / Flex:** data residency is wherever you deploy. We see nothing.

Subprocessors

Subprocessor	Purpose · scope
None (customer content)	No subprocessor receives customer content under our standard SaaS deployment. The platform is operated end-to-end on our infrastructure in the UK.
Auth0 (Okta)	Identity provider for SaaS authentication. Email address, profile info, MFA factor metadata. No customer content.
Cloudflare	Edge proxy, DDoS protection, DNS. TLS-terminated traffic transits Cloudflare; we re-terminate at our origin.
Stripe	Payments. Billing email, project IDs, line items. No customer content.
OVH Cloud (UK)	CI backup destination. Pipeline metadata only.
OpenAI / Anthropic	**Only if configured.** Customer prompts and document chunks transit only when the deployment is configured to use a remote model and a user selects it.

Retention & deletion

Customer content is retained until the customer deletes the workspace. Deletion is cryptographic — once the workspace's wrapped keys are removed from `workspaceusers` and the deployment key (where applicable) is rotated, the underlying ciphertext is irrecoverable even by

PART 9

Incident & support

What happens if something goes wrong — and how we behave when it does.

9.1 Incident response & support access

How we classify incidents, who we notify, and how we handle support that needs to touch production.

ARBI is a small, founder-led company. Security incidents are handled directly by the founders, as the top priority — there is no support tier between you and the people who built the system. We don't claim a 24/7 on-call rota we couldn't staff; we claim fast, honest, founder-level response and a hard commitment to tell you the truth quickly.

Severity classification

Severity	Definition · response
SEV-1	Confirmed data exposure, sustained outage, or active exploitation. Treated as an all-hands emergency; affected customers are contacted directly and kept informed as we work the incident.
SEV-2	Degraded service or suspected (but unconfirmed) data exposure. Prioritised immediately; affected customers kept informed.
SEV-3	Limited-scope issue affecting a subset of users or a single feature. Addressed on a near-term basis; post-mortem if customer-impacting.
SEV-4	Cosmetic, internal-only, or workaround-available. Tracked in the normal engineering queue.

Customer notification

For any SEV-1 or SEV-2 that involves customer data or sustained service impact, we notify affected customers **without undue delay** — the GDPR phrasing — and in any event within 72 hours of confirmation, with a written summary of what happened, what we know, and what we are doing. Where the customer is the controller for personal data potentially affected, we provide the materials needed for the customer's own breach-notification analysis.

Post-mortems

For any confirmed SEV-1 or SEV-2, we give affected customers a written post-mortem — what happened, the impact, and our remediation — shared under NDA. We do not publish post-mortems openly today. PLAN

Vulnerability disclosure

Reports to security@arbi.city. We acknowledge reports promptly and prioritise fixes by severity. We don't run a paid bug-bounty programme, but we credit researchers who disclose responsibly.

Support access

APPENDICES

Reference

A quick reference for the cryptographic primitives, a glossary, and revision history.

A. The crypto, in one table

Every algorithm, key size and library we depend on, with the section in this whitepaper where the construction is explained.

Use case	Primitive
Workspace content encryption	XSalsa20-Poly1305 (NaCl SecretBox), 256-bit symmetric key, 192-bit random nonce per ciphertext. See §3.1.
Workspace key wrapping	NaCl SealedBox = ephemeral X25519 ECDH + XSalsa20-Poly1305. Anonymous, authenticated. See §3.2.
Password → keypair derivation	Argon2id (libsodium `crypto_pwhash`), parameters: libsodium INTERACTIVE preset (2 ops, 64 MiB memory), 32-byte seed output. Salt = SHA-256(email|deployment_domain)[:16]. See §4.1.
Login challenge signing	Ed25519 signature over `email|unix_timestamp`. See §4.2.
Session ephemeral keypair	X25519 generated per login. Public half in JWT (`sk` claim). Private half wrapped with SHA-256(JWT signature), stored in Redis. See §3.3.
JWT signing	RS256 with TPM-resident RSA key where available (single-host on-prem), otherwise Docker-secret RSA key with Redis-backed JWKS distribution. See §4.2 and §6.4.
Direct message encryption	NaCl box = X25519 ECDH + XSalsa20-Poly1305, encrypted to recipient's long-term X25519 public key. See §4.3.
Browser-side signing-key wrapping	AES-256-GCM via WebCrypto. Wrapping key is a non-extractable WebCrypto `CryptoKey`. 96-bit random IV per write. See §7.2.
Content hashing (dedup)	HMAC-SHA256 keyed with the workspace key. Cross-workspace dedup leakage impossible. See §3.1 fact strip.
TLS in transit	TLS 1.2+ / 1.3 preferred. Cloudflare edge + Let's Encrypt origin. mTLS on internal & admin planes. See §6.2.
Database transport	SCRAM-SHA-256 + TLS with client-cert verification (ansible-cluster Postgres). See §6.2.
PRNG	libsodium `randombytes_buf` (browser: `crypto.getRandomValues`; server: `/dev/urandom`). Used for keys, nonces, session IDs, `jti`.
Hashing	SHA-256 for content hashing, salt derivation, JWT-signature wrapping derivation.

B. Glossary

The vocabulary used throughout this whitepaper, defined briefly.

At rest
The state of data when no member is actively transacting with it. ARBI's promise is specifically about this state. See §1.1.

Workspace key
The 256-bit symmetric key that encrypts every confidential field of a single workspace. One key per workspace.

Wrapped key
A workspace key encrypted to a specific member's X25519 public key using NaCl SealedBox. One per (member, workspace).

SealedBox
NaCl's anonymous public-key encryption construction. Ephemeral X25519 sender keypair + XSalsa20-Poly1305 under the derived shared secret. Recipient identity proven cryptographically; sender identity not.

Session key (server)
Ephemeral X25519 keypair generated on each login. Public half travels in the JWT; private half is wrapped with SHA-256(JWT signature) and stored in Redis.

Two-factor session
The model where the session secret is split across the client (JWT signature) and the server (wrapped key in Redis), requiring both to derive an unwrap.

Confidential fields
The hardcoded list of payload field names that are workspace-key-encrypted at rest and stripped from telemetry. Defined in `src/constants.py`.

RLS
PostgreSQL Row-Level Security. Database-enforced filtering of returnable rows by the current session's user GUC. Defence-in-depth alongside encryption.

Workspace
The matter-shaped unit of content and collaboration. Has its own encryption key, its own member list, and its own role assignments.

Direct message
One-to-one E2E-encrypted message between two ARBI users. Independent of workspace membership.

Agent
A persistent, autonomous user-shaped entity owned by a real user. Has its own credentials and its own workspace memberships. Bound by the same RLS and the same agent-token allow-list of sensitive endpoints.

JIT (just-in-time) access
Access we request, scoped to a specific operation and bounded in time, when a support task

C. Contact & revision history

Who to talk to about what, and how this document has changed over time.

Security contact

For vulnerability disclosure, security questionnaire responses, DPA negotiation, audit-evidence requests and incident escalation:

security@arbi.city
We respond to security reports as a priority.
Tel: +44 203 105 0211 (business hours, UK)

Mailing address

Arbitration City Ltd
Juxon House · 100 St. Paul's Churchyard
London EC4M 8BU · United Kingdom
Company number: registered in England and Wales

Revision history

Version	Date · changes
v1.0	May 2026 — Initial public release. Replaces the internal A1 Security Overview / A2 Security Whitepaper customer-pack documents with this code-grounded, gap-disclosed version. Authored against the May 2026 codebase.

Companion documents

- **Security One-Pager (v1.0)** — the printable executive summary. arbicity.com/ARBI-Security-Onepager.pdf
- **ARBI Docs User Guide** — how the product actually works from a user's perspective. arbicity.com/ARBI-Docs-User-Guide.pdf
- **Data Processing Addendum** — available on request from security@arbi.city.
- **Subprocessor list (current)** — see §8.2; the live version is mirrored on our security page.

A STANDING INVITATION If a clause in this whitepaper does not say what your procurement, security or compliance team needs it to say, write to us. We update this document on a rolling cadence and would rather rephrase a section once than have ten customers each work around the same ambiguity.