POSTSPACE E2EE Ver. 2.1 Specification

Prototype implementation — **core E2EE** items up to 21.

[Overview]

- A **PIN** is stretched into an **E2EE key (Master Key)** and shared between sender and recipient via a link to realise end-to-end encryption.
- At no point is the Master Key present on the server **(server-side absence).** Wherever it is stored (sender device, link, recipient device) it is always encrypted **(wrapping).**
- The Master Key exists only temporarily in device memory **(virtual key)**. Moreover, the fundamental elements of the architecture—PIN and Master Key—are never used directly as cryptographic keys; after use they are wiped from memory **(memory-layer zeroisation).**

[Basic Flow]

1. **User registration (sender)**
    - A system-generated PIN is converted into the **Master Key**.
    - The Master Key is wrapped with the **User Key** to produce a **User–Master Key**, which is saved on the sender's device.
2. **Upload (sender)**
    - The Master Key (unwrapped from the User–Master Key) and an **Upload Key** derive a **File Key**, which encrypts the file.
3. **Registration-link issuance (sender)**
    - The Master Key is wrapped with a **Link Key** to create a **Link–Master Key**, which is issued inside the registration link.
4. **Registration-link access (recipient)**
    - The recipient unwraps the Link–Master Key with the Link Key to obtain the Master Key.
    - The Master Key is then wrapped with a **Device Key** to create a **Device–Master Key**, saved on the recipient's device.
5. **Download (recipient)**
    - The Master Key (unwrapped from the Device–Master Key) and the Upload Key re-derive the File Key, which decrypts the file. PSGPT_7_E2EE_ver2_1

[Key Points]

- The Master Key (256 bit) derived from a stretched 8-digit numeric PIN has sufficient entropy; an 8-digit PIN is acceptable.
- To guarantee true E2EE, the Master Key is generated and held **only on the devices**; all server processes operate without it.
- The File Key is the final key used to encrypt files. The Master Key serves as the root key from which the File Key is derived. Both Master Key and File Key are **virtual keys**—they exist only in memory and are discarded immediately after use.
- All wrapping keys (**KEKs**) are stored in MongoDB; wrapped keys themselves reside only on devices or in links.
- Even if both the database and object storage are breached simultaneously, no plaintext leaks unless the device is also compromised, and vice versa.
- Even if all three components—the database, storage, and device—are compromised simultaneously, damage is limited solely to files within the affected device's scope and their retention period.
- The Master Key is transmitted only once (in the registration link), never in the download link, minimising network-side leakage risk.
- The most realistic remaining risk is accidental mis-sending of the very first registration link. As mitigation, a **two-step delivery** option allows the sender to send a registration link alone, confirm the recipient by phone, then send the download link. For ordinary convenience, this remains optional; for corporate accounts the two-step process can be made mandatory in the paid plan. Corporate accounts (paid) also can configure custom registration periods and file retention periods.

[Key Glossary]

Key	Purpose / Generation	Storage
PIN	8-digit number, generated client-side at user registration; sole material for the Master Key.	Not stored server-side; optionally encrypted and cached client-side.
User Key	Salt for stretching the PIN into the Master Key; also KEK for the **User–Master Key** and for encrypting the PIN backup. Generated by hashing the User ID (DB ObjectId → hex) with HMAC-SHA-256.	users collection
Master Key	256-bit key produced by Argon2id stretching of the PIN. Root for all File Keys. **Virtual** (memory-only).	Not stored.
User–Master Key	Master Key wrapped (AES-256-GCM) with the User Key; saved on the sender's device.	LocalStorage (sender).
File Key	Final key that encrypts each file; derived per upload via HKDF(Master Key, Upload Key). **Virtual**.	Not stored.
Upload Key	Salt to vary the File Key per upload; HMAC-SHA-256 of the Upload ID.	uploads collection.
Device Key	KEK for the **Device–Master Key**; HMAC-SHA-256 of the Device ID.	devices collection.
Device–Master Key	Master Key wrapped with the Device Key; saved on the recipient's device.	LocalStorage (recipient).
Link Key	KEK for the **Link–Master Key**; HMAC-SHA-256 of the Link ID (generated server-side).	links collection.
Link–Master Key	Master Key wrapped with the Link Key; embedded in the registration link code.	In the link only (not in DB).

[Detail]

① PIN generation and hashing (user registration)

- An 8-digit **PIN** is generated automatically on the client; the user never chooses it.
- The PIN is the sole source material for the **Master Key** and is also used for PIN-based login on another device.
- To create a **PIN-authentication key** the PIN is hashed with Argon2id:
  - Salt = 16 bytes CSPRNG (getRandomValues)
  - Parameters = memoryCost 19456, timeCost 2, parallelism 1, hashLength 32
  - The resulting PHC string
    $argon2id$v=19$m=19456,t=2,p=1$<saltB64>$<hashB64>
- PIN is stored in the users collection; the raw PIN is **never** stored.
- A secret pepper is kept in .env.
- Rate-limit: after password auth, 3 consecutive PIN failures → 1-hour lock.

② PIN backup on the sender device

- At login the PIN is encrypted with the **User Key** (AES-256-GCM, CryptoKey extractable:false) and saved to LocalStorage as Base64url.
- After encryption the buffers are zero-filled and set to null.
- The encrypted PIN is deleted on logout, or automatically after **one week**. Create a timestamp key in LocalStorage; when reading, check expiration. If expired, delete along with PIN.
- if expired, PIN auth is required again.

③ Master Key generation

- The PIN is stretched with Argon2id (salt = User Key) to create the 256-bit **Master Key**.
- The Master-Key is used to generate the File-Key and Link-Master-Key. The Master-Ket is virtual key that exists only in memory.

④ User–Master Key registration

- The Master Key is wrapped with the User Key (AES-256-GCM) → **User–Master Key** (Base64url) and stored in the sender's LocalStorage.
- After any crypto operation the key material is zeroised and nulled.
- Delete upon logout; regenerate upon login.
- Overwrite if an existing value is present.
- Retention period: 90 days. Create a timestamp key in LocalStorage; when reading, check expiration. If expired, delete along with the User–Master Key.
- Not stored in DB.
- (IV & TAG stored together)　saved in the users collection.

⑤ Re-wrapping concept

・ The Master Key is re-wrapped for transfer (**Link–Master Key**) and for recipient storage (**Device–Master Key**).

⑥ User Key generation

・ At user registration the **User Key** is
HMAC-SHA-256(key = .env fixed value , msg = "user|<userIdHex>")
・ Stored in the users collection.

⑦ File Key derivation

・ For every upload the **File Key** is derived by
HKDF(Master Key , salt = Upload Key)
・ and used for AES-256-GCM encryption; the File Key is a virtual key, existing only in memory.

⑧ Browser-side zeroisation limits

・ Because Web Crypto CryptoKey objects cannot be overwritten, front-end handling is:
  1. create keys with extractable:false;
  2. after use set key = null (GC eligible).
・ Plaintext or ciphertext buffers *can* be zero-filled; do that before null.

⑨ Upload Key generation

・ HMAC-SHA-256(key = .env fixed value , msg = "upload|<uploadIdHex>")
・ Stored in the uploads collection.

⑩ Link Key generation

・ HMAC-SHA-256(key = .env fixed value , msg = "link|<linkIdHex>")
・ Stored in the links collection.

⑪ Link–Master Key

・ AES-256-GCM(plaintext = Master Key , key = Link Key) → **Link–Master-Key**.
・ Generated by the back-end when issuing a registration link.
・ After encryption the variables are zero-filled/nulled.
・ Not stored in DB; used only inside the link.
・ (IV & TAG) stored in the links collection.

⑫ Registration-link code

・ Base64url(<Link-ID binary> || <Link–Master Key>)
・ No server storage.

⑬ Device Key generation

- HMAC-SHA-256(key = .env fixed value , msg = "device|<deviceIdHex>")
- Stored in the devices collection.

⑭ Device–Master Key registration

- AES-256-GCM(Master Key , key = Device Key) → **Device–Master Key** (Base64url).
- Saved in the recipient's LocalStorage together with the Device ID (hex). Not stored in DB.
- Overwrite if an existing value is present.
- Retention period: 90 days. Create a timestamp key in LocalStorage; when reading, check expiration. If expired, delete along with the Device–Master Key and Device ID.
- No server storage.
- (IV & TAG) stored in the links collection.

⑮ Complete user-registration sequence

1. User ID → User Key stored
2. PIN stretched to Master Key → wrapped to User–Master Key
3. Both PIN (encrypted) and User–Master Key saved to LocalStorage. (overwriting if the device already exists).

⑯ Complete upload sequence (sender)

1. Upload ID → Upload Key stored
2. User–Master Key unwrapped
3. File Key derived → File encrypted.

⑰ Complete registration-link issuance (sender)

1. Link ID → Link Key stored
2. User–Master Key unwrapped → Master Key
3. Master Key wrapped with Link Key
4. Link–Master Key → registration-link code issued.

⑱ Complete registration-link handling (recipient)

1. Device ID → Device Key stored
2. Link code yields Link–Master Key unwrapped → Master Key
3. Master Key Wrapped with Device Key → Device–Master Key saved (overwriting if the device already exists).

⑲ Download sequence (recipient)

1. Device–Master Key unwrapped → Master Key
2. File Key derived → File decrypted.

⑳ Download-link issuance

1. 16-byte random via getRandomValues; **DL code** =
2. Base64url(<Link-ID binary> || <random>), dummy-padded to a fixed **110 characters** so link type cannot be inferred.
3. Contains no Master Key and is not stored in DB.

21 Login from another device

・ When a sender logs in from a different device, they must first pass **password authentication and then PIN authentication** before access is granted.

22 Rate-limit for PIN authentication

・ Because password authentication is already rate-limited, PIN authentication adds a light throttle: **maximum 3 consecutive failures per hour** when logging in from another device.

23 Effect of a PIN reset

・ If the sender resets their PIN, **all files and all issued links are deleted**. Recipients remain, but **all devices are reset**.

24 Recommendation to reset PIN frequently

・ The service stores files only temporarily, so the practical impact of a PIN reset is small. For operations that prioritise security over convenience, frequent PIN resets are recommended and assumed.

25 Warning shown when a PIN is issued

・ “Please record this PIN or store it in a backup file. If you lose the PIN it cannot be re-issued and you will be unable to log in. Keep it in a safe place.[Store PIN in backup file] [Next]”

26 Mandatory memory-wipe order for cryptographic variables

・ During any encryption/decryption the following memory-wipe sequence is mandatory:
  1. **Zero overwrite** ciphertext = "" (Base64 / JSON) -or- ciphertextBytes.fill(0) (Uint8Array)
  2. *(Front-end only, if needed)* structuredClone for asynchronous safety
  3. **Null assignment** ciphertext = null (marks object for GC)

  * Reversing the order (null first, then overwrite) fails to wipe the data.

27 Handling of CryptoKey objects in the browser

・ With Web Crypto API the CryptoKey itself cannot be zero-filled. Client-side rules:
  1. Always create keys with extractable: false.

2. After use set key = null so they become garbage-collectable.
   Plaintext and ciphertext buffers **can** be zero-filled and must be wiped as in item 26.

28 Residual-risk testing and supported browsers

・ The wipes above cannot eliminate browser-implementation variance completely, but they reduce the practical risk of PIN or Master Key leakage to a negligible level.
Run memory-analysis tests per browser (E2EE unit test + Puppeteer execution → heap-snapshot diff) to confirm no residual keys. Classify browsers into “recommended / not recommended” and present that list to users.

29 Protection against LocalStorage & XSS

・ Only the recipient-name field allows free text; all other inputs are file uploads.
・ Apply sanitising, HTML escaping, and use textContent/innerText instead of innerHTML.
・ Consider ZIP-wrapping pdf/txt/html/svg/xml uploads.
・ On download set HTTP headers:
Content-Security-Policy: default-src 'self'; script-src 'self' and
Content-Disposition: attachment.

30 Precautions for Office-format files

・ Because Edge may auto-open Office files and execute macros:
・ On upload record the original filename.
・ On download set headers, e.g.
Content-Disposition: attachment; filename="filename.xlsm"
Content-Type: application/vnd.ms-excel.sheet.macroEnabled.12
(filename is the sanitised original).
・ Display a warning such as “This file contains macros. Open only if you trust it.”

31 Separation of secrets in production

・ To prevent chain compromise if the application EC2 is breached: S3 access is controlled by **IAM roles**.
・ MongoDB secrets are managed by **AWS Secrets Manager / KMS**.

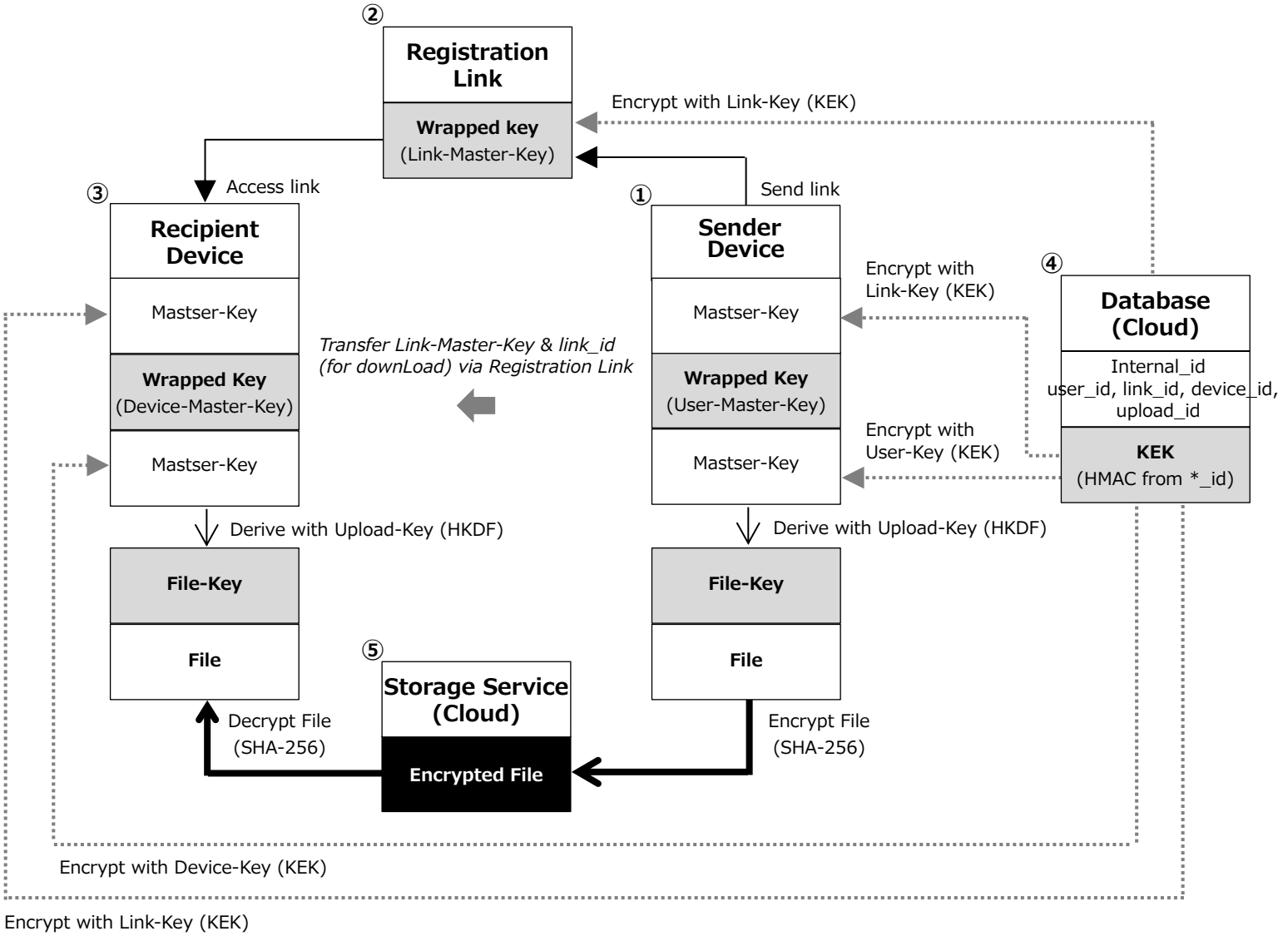
32 KMS operation and key-rotation policy

・ When migrating to KMS, the **key names and formats must remain identical** between .dev and production.
・ After rotation, old keys remain marked AWSPREVIOUS; define a **grace period**, monitor that the application has switched to the new key, then delete the old version.
・ **User-Key HMAC keys**: KMS rotation optional, recommended once per year.
・ **Other KEKs**: manual rotation every 90–180 days according to the runbook

(https://github.com/postspace/pse2ee/runbook/key-rotation/).

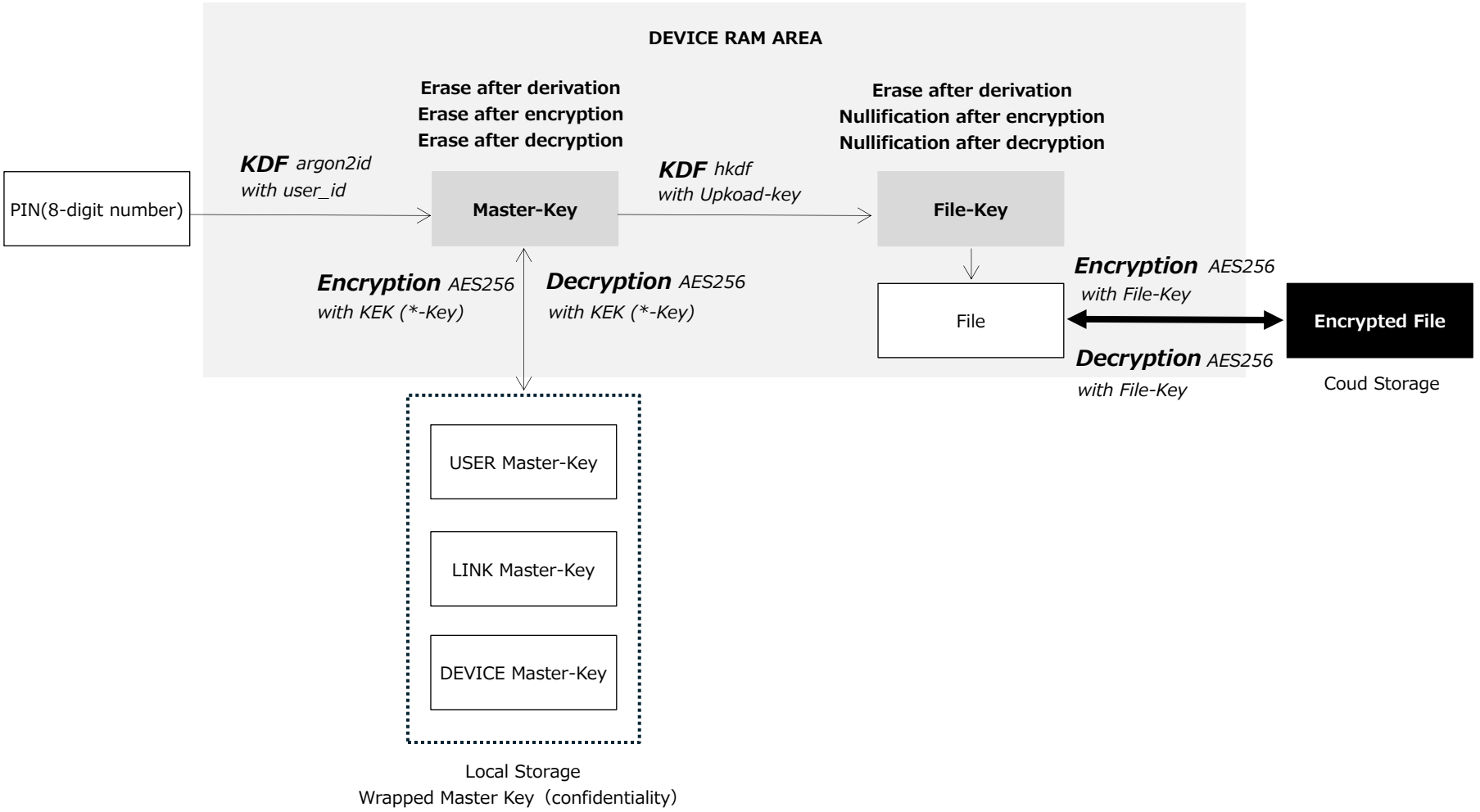
Please refer to the attached document for charts and diagrams.

E2EE (Master-Key) Conceptual Diagram



E2EE (Master-Key) Virtualization Conceptual Diagram

ALGORITHM	INPUT	PARAMETER	OUTPUT
KDF（argon2id, HKDF)	Input key Material	Salt	Derived key
Encryption（AES256-GCM)	Plaintext	Encryption Key	Ciphertext



Flow for Generating Each Cryptographic Key from DB Internal Identifiers

Key generation process	STEP ① Input material for Algorithm ②(DB internal iD)	STEP② Algorithm that generates the input value	STEP ③ Result of Algorithm ② (derived key)	STEP ④ Algorithm that generates the key	STEP ⑤ Result encrypted by KEK ③ (cipher-text)	STEP ⑥ Result derived with Salt ③ (derived key)	Role of the key
Master-Key	user_id	argon2id	Master-Key	—	—	—	(1) Plaintext when encrypting **registration-link code** (2) Key material when deriving **File Key** (final encryption key)
User-Master-Key	user_id	HMAC	User Key **(KEK)**	SHA-256-GCM	User-Master-Key	—	Master-key wraped on **sender device**
Link-Master-Key	link_id	HMAC	Link Key **(KEK)**	SHA-256-GCM	Link-Master-Key	—	Master-key wraped inside **registration-link code**
Device-Master-Key	device_id	HMAC	Device Key **(KEK)**	SHA-256-GCM	Device-Master-Key	—	Master-key wraped on **recipient device**
Upload-Key	upload_id	HMAC	Upload Key **(Salt)**	HKDF	—	File-Key	Salt for **File-key** (final encryption key) derivation

All E2EE Key Specification Table

Key category	Key material (input)	Algorithm that derives the key	Salt / Key	Unit of generation	Triggering event	Generation place	Storage place	Role of the key (output)	Encryption target / plaintext	Encrypted data / ciphertext (derived key)	Where ciphertext is stored (DB)	Where ciphertext is stored (device)	Purpose of encryption / key-derivation
PIN	Random 8-digit numeral	—	—	User	User registration	Front-end (sender)	*None* (memory → zeroised)	Source material of Master Key	—	—	—	—	—
PIN-authentication key	PIN	Argon2id	getRandomValues(16 B) *1	User	Login	Front-end (sender)	DB / users (salt stored)	Verifies entered PIN	—	—	—	—	Device roaming
Master Key	PIN	Argon2id	User Key	User	User registration	Front-end (sender & recipient)	*None* (memory → zeroised)	Source material of File Key	—	—	—	—	—
File Key	Master Key	HKDF *2	Upload Key	Upload	Upload	Front-end (sender & recipient)	*None* (memory)	Data-encryption key *3	File	Encrypted file	S3 *4	—	Storage confidentiality
Upload Key	Upload ID	HMAC-SHA-256 *2 (fixed key)	—	Upload	Upload	Front-end (sender)	DB / uploads	Salt for File-Key derivation	(Master Key)	(File Key)	—	*None* (memory → zeroised)	Per-upload variation
User Key	User ID	HMAC-SHA-256 *2 (fixed key)	—	User	User registration	Front-end (sender)	DB / users (IV,TAG)	Salt for Master-Key derivation	(PIN)	(Master Key)	—	*None* (memory → zeroised)	Key strengthening
								PIN-backup KEK *3	PIN	Encrypted PIN	—	Sender LocalStorage	Prevent PIN loss
								User–Master-Key KEK *3	Master Key	User–Master Key (enc. Master)	—	Sender LocalStorage	Device confidentiality
Link Key	Link ID	HMAC-SHA-256 *2 (fixed key)	—	Link	Registration-link issue	Back-end (API)	DB / links (IV,TAG)	Link–Master-Key KEK *3	Master Key	Link–Master Key (enc. Master)	—	Registration-link code	Network confidentiality
Device Key	Device ID	HMAC-SHA-256 *2 (fixed key)	—	Device	Registration / DL-link access	Front-end (recipient)	DB / devices (IV,TAG)	Device–Master-Key KEK *3	Master Key	Device–Master Key (enc. Master)	—	Recipient LocalStorage	Device confidentiality

*1 The PIN-authentication key is stored as an Argon2id hash for device roaming; its salt is a random value different from the User ID. A secret pepper is stored in the application .env.

*2 The HMAC key is an individual 32-byte random value kept in .env; migrating to Secrets Manager + KMS for production is under evaluation (dev/stg → .env, prod → Secrets Manager/KMS).

*3 All encryption operations use **AES-256-GCM**.

*4 In the prototype, encrypted files are stored in the local uploads folder; production will use AWS S3.

Cryptography Terminology

Plaintext
Data that needs to be protected through encryption. It is the input to the encryption process. Even when in binary format, it is still referred to as "plaintext."

Ciphertext
Data that has been protected using an encryption key. It is the output of the encryption process. Even in binary format, it is referred to as "ciphertext."

Encryption Key
A secret piece of information used in the encryption process. It is part of the input to the encryption algorithm and is specified independently of the plaintext. Encrypting the same plaintext with the same algorithm and the same encryption key (assuming identical randomized parameters such as IV) results in the same ciphertext. Using a different key results in a different ciphertext. Encryption keys include symmetric keys (used in symmetric-key encryption) and public/private key pairs (used in public-key encryption).

Encryption
The process of converting plaintext into ciphertext using an encryption key.

Decryption
The process of converting ciphertext back into plaintext using an encryption key.

Key Derivation
The process of generating a new secret key from an existing one using a Key Derivation Function **(KDF)**. It is used to derive keys for different purposes (e.g., password-based authentication keys), to adjust key length, or to isolate keys for specific applications. Key derivation is a **non-reversible** transformation and is fundamentally **different from encryption**, which is designed to be reversible through decryption.

Key Material
The input value provided to a KDF.

Salt
An auxiliary input to a KDF. Even with the same key material, different salts will result in different derived keys. Salts are used to protect against dictionary attacks and rainbow table attacks.

Derived Key
The output value generated by a KDF.

Web Crypto API
The standard JavaScript API for encryption and key derivation in web browsers. Unlike general-purpose JavaScript, all input/output values for encryption and key derivation **must be binary data**. If the output must be handled as a string (e.g., for display or storage in Base64 or Hex), explicit conversion between binary and string is required before or after encryption. Another key difference from regular JavaScript is that the memory content of a encryption Key **(CryptoKey object) cannot be explicitly zeroed out** (e.g., using fill(0)), meaning secure memory management cannot be enforced via JavaScript. While a CryptoKey can be dereferenced (e.g., set to null), the timing of actual memory release is browser-dependent and cannot be strictly controlled by the programmer. In contrast, **plaintext and ciphertext data can be explicitly zeroed out**.

Why do we need E2EE?

From a risk-management perspective, we compare (1) on-premise physical servers, (2) cloud services, and (3) cloud + E2EE, and explain the effectiveness and significance of end-to-end encryption (E2EE).

(1) Physical servers

Even minor vulnerabilities lead directly to massive data breaches.

Because a small flaw in the server OS or application can let an attacker obtain complete system information—such as OS-level privileges or memory dumps—large-scale data breaches occur frequently.

(2) Cloud services

Human error or insider wrongdoing in cloud administration leaves residual leakage risk.

Even if an application server is totally compromised, risk can be mitigated by isolating data in separate cloud services. Strict, region- and role-based control of API keys and IAM (e.g., KMS, cross-region separation) further reduces exposure. In reality, however, misconfigurations and insider abuse continue to cause leaks. Modern cloud environments are increasingly serverless and account-distributed; log aggregation and threat detection are delayed, and many breaches are discovered only after stolen data are misused. Under the de-facto Shared Responsibility Model (SRM) in today's contracts, configuration errors and insider incidents fall on the customer. As a result, operational burden and risk are growing. High-profile examples include Capital One (2019) and Optus (2022), where millions of customer records were exposed through human error—costing the companies hundreds of millions of dollars.

(3) Cloud + E2EE

A design that stores neither keys nor plaintext in the cloud, dramatically lowering the likelihood of large-scale leaks.

E2EE is an endpoint-centric encryption architecture. Even when an application is breached and cloud administration fails, encryption keys remain on endpoints, and plaintext data never reside in the cloud, preventing disclosure. Some metadata (in POSTSPACE: recipient label, timestamp, etc.) may still be exposed, but the actual file contents remain protected. Leakage of real data can occur only if all three conditions happen simultaneously:

1. Application compromise
2. Cloud misconfiguration
3. Endpoint key theft

Even then, damage is limited to data currently stored on the affected endpoint (three days' worth in POSTSPACE). While E2EE is commonplace in some mobile messaging apps, adoption on PCs has lagged due to the complexity of key management. POSTSPACE targets business use, balancing usability and security with a device-agnostic key-management design to promote wider E2EE adoption.

Summary

It is unrealistic to drive the individual risks of **1. application compromise, 2. cloud misconfiguration, and 3. large-scale endpoint malware infection** to zero. **The probability that all three occur simultaneously is extremely low. E2EE blocks real data leakage in all other scenarios**, thus markedly increasing trust in cloud services. By adopting E2EE, even if servers or clouds are compromised the keys remain protected and the actual data stay confidential. Residual risks can be confined locally through conventional endpoint defenses, bringing the likelihood of a large-scale breach down to a level that is operationally negligible.