ChatGPT

Building a Cloud File Uploader with Hash-Based Access

Overview & Key Features

Building this app involves a **front-end chat interface** and a **back-end service** to handle file uploads and storage. The goal is a web app that lets users upload files of any size to Backblaze B2 cloud storage and retrieve them via a unique hash (like an ID) generated on upload. Key features include:

- **Unlimited File Uploads:** Support for very large files by using chunked/multipart uploads.
- **Unique Hash IDs:** Each uploaded file gets a unique hash identifier (e.g. a random or content-based hash) that serves as its retrieval key [1].
- **Hash-Based Grouping:** Users can designate one file's hash as a *parent hash* to group related files. Visiting the parent hash URL shows a **gallery view** of all linked files.
- **Chatbox UI:** The web interface is styled like a chat (à la ChatGPT). The chat panel (center of the screen) allows text input of commands or hash queries, and displays responses (file links, gallery previews, etc.). An **Upload** button allows selecting files to send.
- **Auto-Tagging & Metadata:** On upload, users describe the file for tagging. The system can also use AI to auto-generate tags or descriptions (e.g. image content labels, text extraction) [2]. These tags aid in search and sorting.
- **Gallery & Sorting:** If a parent hash is accessed, the app shows a *gallery view* of file previews (images, videos, documents, etc.) with state-of-the-art sorting and filtering. Users can sort by file type, upload date, tags, or even use AI to group similar content.

System Architecture

Front-End (Chat Interface & Uploader)

- **Chat Interface:** A single-page web app (could be built with React, Vue, or any modern framework) presents a chat-like UI. The center panel is a conversation view. This interface welcomes user input (text or file). It provides an intuitive experience – for example, the user can type a hash into the chat, and the app will respond with the corresponding content or gallery.
- **Upload Widget:** An upload button or drag-and-drop zone is integrated into the chat UI (possibly as part of a message composer). When the user selects a file, the front-end first contacts the back-end to get an authorized upload URL or token for Backblaze B2. With that, it uploads the file directly *from the browser to B2* without routing through your own server [3]. This is crucial for large files, as it avoids server timeouts and memory issues.
- **Progress & Feedback:** The front-end should show upload progress (especially for large files). After a successful upload, it receives the unique hash ID and displays it to the user (e.g. as a chat message: "Uploaded *filename* – Access it via hash: **Xyz123**"). It may also confirm any tags the user provided or suggest additional auto-tags.

Back-End (Server & Database)

- **REST API or WebSocket:** The back-end (using Node.js/Express, Python/FastAPI, or similar) provides endpoints for: requesting an upload URL, confirming an upload, retrieving file metadata, and listing files by parent hash. A WebSocket or SSE could be used to stream status updates or allow the chat UI to feel real-time.
- **Backblaze B2 Integration:** The server holds the B2 credentials (Application Key and Key ID) and never exposes them to the client. When an upload is initiated, the server either generates an **S3 presigned URL** or calls the B2 Native API to get an **upload URL & auth token** [4]. These are sent to the client for direct upload. For B2's native API, the client must include the file in an HTTP POST along with the auth token and file SHA1 hash for integrity [5]. (B2 requires a SHA-1 checksum of the file; the app can compute this in JS via Web Crypto API.)
- **Handling Large Files:** For files >5 GB, the app must use B2's multipart upload. Backblaze's S3-compatible API supports standard S3 multi-part upload, and the native API provides "Start Large File" and "Upload Part" calls [6] [7]. The server can orchestrate this: e.g., break the file into chunks (perhaps 100MB each), get upload URLs for each part, and have the client upload each part with its SHA1. Once all parts are uploaded, a "Finish Large File" call assembles them [8]. This allows **virtually unlimited file size** support.
- **Database & Metadata:** A database is used to store file metadata:
- **Hash ID** – primary key to identify the file (could be a random GUID or a content hash like SHA-256 of the file content for deduplication). This is what users will share.
- **File Info** – original filename, size, type/MIME, upload timestamp, tags (both user-provided and auto-generated).
- **Storage Pointer** – info to retrieve file from B2 (e.g. B2 file ID or the bucket and object name).
- **Parent Hash** – an optional field referencing another file's hash if this file is part of a group. If set, it means this file belongs to the collection identified by the parent hash.
- **Business Logic:** After a file upload finishes, the back-end:
- Generates the unique hash for the file. This might be done by hashing the file content to produce a content-addressed ID (similar to IPFS CIDs, which use a cryptographic hash of the content [1]) or by generating a UUID/random string. Content hashing ensures identical files map to the same ID (useful for avoiding duplicates), whereas random IDs ensure every upload gets a new identifier. In either case, the hash should be long enough to avoid collisions (e.g. 128+ bits if random or using a strong hash function).
- Stores the file record in the database with any tags. If the user provided a description/tags, save those. If auto-tagging is enabled, the server could asynchronously call AI services (like image recognition, text analysis, etc.) to get tags. For example, an image upload could be sent to an AI vision API which returns tags (cat, outdoor, sunset, etc.) [2], and a PDF could be run through OCR or NLP for keywords. These tags help with the gallery sorting/search later.
- Returns the unique hash to the front-end, which displays it to the user.

Cloud Storage (Backblaze B2)

- **Bucket Setup:** All files reside in a B2 bucket. You can store them with the hash as part of the filename for easy reference (e.g. `<hash>__originalFilename` or just the hash as name, since the DB can map to original name). The bucket can be public, allowing direct file download via a URL, or private (then your server would need to generate temporary download URLs or proxy the download).
- **Direct Download:** To serve files via the hash, you have two approaches:

- **Via B2 Public URL:** If the bucket is public and you name objects by their hash, users can fetch `https://f000.backblazeb2.com/file/YourBucket/<hash>` directly. However, since our app might use hashes to refer to either single files or galleries, it's better to funnel all access through our app's front-end.
- **Via App Server:** The client requests `https://yourapp.com/files/<hash>` (or simply enters the hash in chat). The server looks up the hash in DB to find the B2 file reference. If it's a single file, the server can redirect to the B2 download URL or stream the file. If the hash corresponds to a parent (collection), the server returns metadata for all files with that parent for the front-end to display as a gallery.

File Retrieval and Linking

Single File Access by Hash

When a user inputs a hash into the chat interface (or navigates to a hash-specific link), the app will determine if it's a file or a collection: - **Lookup:** The front-end sends a request to the back-end (e.g. `/api/metadata/<hash>`). The server finds the record. If `parentHash` is not set for this record (or the record is not marked as a collection container), it's a single file. - **Response:** The server responds with file metadata and either a direct download link or a short HTML/JSON snippet for the front-end to render. In the chat UI, the app could display a message like "**File:** *filename.ext* (*123 KB*) – [Download] [Preview]". For previewable types (images, text, maybe PDFs), the UI can show a thumbnail or inline preview. (For example, images can be displayed directly in the chat conversation as if the "assistant" posted the image.)

Gallery View for Parent Hash

If the hash corresponds to a *parent collection* (i.e., one or more files have their `parentHash` equal to this hash): - **Listing Files:** The back-end finds all file records where `parentHash = <that hash>`. It then returns a list of metadata (filenames, their own hashes, thumbnails, tags, etc.). - **Gallery UI:** The front-end will render a gallery page or an embedded gallery component in the chat. This gallery should beautifully display each file: - **Previews:** Depending on file type, show an appropriate preview. Images: a thumbnail or scaled-down image. Videos: a thumbnail (first frame) or an icon with play button. Audio: perhaps an icon or a small audio player. Documents: an icon or preview (e.g. PDF first page or text snippet). For miscellaneous binary files, use a generic icon. - **Metadata Overlay:** Each item can show its filename or icon, maybe the size and an option to download. Hover effects or click could show more details (full title, tags). - **Sorting & Filtering:** The gallery interface can include controls to sort the items. Basic sorts: by upload date, name, size, file type. With tags available, users can also filter by tag (e.g. only show files tagged "invoice" or "cat"). "State-of-the-art sorting" could imply intelligent sorting – for instance, *AI-driven sorting*: if the collection is large, the system might cluster similar images together or highlight frequently accessed files. You could implement a **search bar** for the gallery that uses the tags and file content (for example, a user can type "sunset" to quickly filter images auto-tagged as sunset). - **AI Canvas-like Interaction:** Taking inspiration from OpenAI's Canvas (which allows side-by-side interaction with content and an AI assistant) – you could allow the user to click an "Assist" button in the gallery. This might let them ask questions in the chat about the collection (e.g. "Summarize the documents in this collection" or "Find the image with a cat and a dog together"). While this is an advanced feature, designing the UI with a chat + content pane in mind could future-proof the app. Essentially, the **chat interface and gallery** live on the same page, so the user can seamlessly switch between conversing (asking the system to do something, like sort or tag differently) and interacting with the file previews.

Implementation Steps

1. Setting Up Backblaze B2 for Direct Uploads

- **Create Bucket:** Make a B2 bucket (public or private). If private, configure your app to generate download auth tokens as needed.
- **CORS Configuration:** Enable CORS on the bucket to allow your web app's domain to upload directly. (Backblaze's documentation provides a JSON policy to permit browser uploads from any origin or specific origins [9].)
- **App Keys:** Generate a B2 Application Key with *write* access to the bucket. These credentials will be stored on your server for initiating uploads.

2. Back-End API Endpoints

Implement server endpoints: - `POST /api/get-upload-url` (or integrate into a single `/upload` flow): Server reads a request (which may include desired filename or content type) and obtains an upload URL. - **Option A: B2 Native API** – call `b2_get_upload_url` to get `<uploadUrl, authToken>` [4], then send these to the client. The client will do an HTTP POST to that URL with file bytes and an `Authorization: authToken` header. - **Option B: S3 Presigned URL** – use B2's S3 API to generate a presigned URL for `PUT`. The server could use AWS SDK (configured for Backblaze endpoint) to presign an `putObject` request for the target object name [3]. Return this URL to client; the client does an HTTP PUT with the file. - The response also includes the **file hash ID** that the client should use after upload (if pre-generated). If using B2 Native, you might generate the hash *after* upload on the server side when you confirm success. - `POST /api/confirm-upload`: (If needed) After the client uploads to B2, it calls this to tell the server the upload is done (along with any returned file info from B2, such as fileId or etag). The server then finalizes the DB record: storing the hash, linking to B2 fileId, etc. If using presigned URL, the server might generate the hash before and use it in the object name, so confirmation might simply allow storing additional metadata. - `GET /api/metadata/{hash}`: Returns metadata about the item identified by `hash`. This includes whether it's a collection or single file, the file name, size, tags, and perhaps a list of child files if it's a parent. - `GET /api/download/{hash}`: (Optional if using direct links) If you want to stream through your server or count downloads, this endpoint fetches the file from B2 (using fileId or name) and pipes it to the response. Alternatively, it can redirect to the B2 file URL. - `GET /api/list/{parentHash}`: Returns a list of files (metadata) linked under a given parent hash, for gallery views. (This could also be merged with the metadata call if that returns children when it's a parent.)

The server should also enforce any security (if you want private datasets or user accounts), but since the prompt implies an open, anonymous system ("anon, random" usage), likely all files are accessible via knowing the hash (like a secret link).

3. Unique Hash Generation

Decide on a method for generating the hash IDs: - **Content-Based Hash:** You can compute a SHA-256 or SHA-1 of the file content. This can be done in the browser (for smaller files) or on the server. IPFS uses this approach to create a Content ID [1]. It ensures the same file content yields the same hash, which is good for deduplication, but it means if two different users upload identical files, they'd get the same hash (which may or may not be desired). Also, computing a SHA-256 in browser for multi-GB files is heavy; doing it stream-wise on the server might be better. - **Random Unique ID:** Alternatively, generate a random UUID or

a NanoID/ULID. This is simpler and ensures uniqueness as long as you check for collisions in the DB. Random IDs avoid any privacy issues (no information leaked from content) and allow duplicate files to have separate hashes if needed (since the focus is on unique labeling for training data, you might actually want separate entries even if content repeats, if they have different tags). - **Hash Format:** The hash can be a short alphanumeric string. For example, a UUIDv4 could be base58 or base62 encoded to make it shorter. Aim for an ID that users can easily copy. You could also use a **two-part code**: one part derived from content and one random, to get benefits of both uniqueness and deduping.

When a file is uploaded, the server creates the hash and returns it. If the user wants to **link files into a collection**, the simplest approach is: they take the hash of one file (intended parent) and during upload of another file, provide that as the "parent" field. The server then sets the new file's `parentHash` to that value in the DB. (You could also allow after-the-fact grouping: e.g. an API call to update a file's parent hash, or a way in the UI to "merge" two files into one collection by choosing one's hash as parent for the other.)

4. Auto-Tagging Implementation

Incorporating auto-tagging will significantly enhance the dataset's usefulness. Here's how to do it: - **User Input Tags:** The upload form can ask the user for a description or tags for the file. These are saved as metadata. - **AI Tagging Services:** For each file, run automatic tagging in the background: - *Images:* Use a Computer Vision API (like Azure Cognitive Services, Google Vision, or an open-source model) to get labels and descriptions of the image content. For example, Vision APIs can return objects detected, scene descriptions, and even OCR text from images [10] [11] . - *Video:* Extract key frames and run image tagging on them, or use video intelligence APIs to detect scenes, objects, or spoken words (if any audio). - *Audio:* Use speech-to-text to transcribe, then extract keywords. - *Documents (PDF, text):* Use NLP to find key phrases or categorize the text subject. - **Metadata Storage:** Merge the auto-tags with user tags in the database. You might store them in a text array column for easy full-text search. The tags will be used in the gallery for filtering/search, and possibly for **smart grouping** (e.g., you could automatically group files with the same tag under a suggested collection). - **Feedback to User:** The chat UI can even display the auto-tags after upload as a message: "Auto-tags: cat, outdoor, sunset". This confirms to the user how the file was labeled. If using AI in the interface, you could allow the user to refine these via chat: "Remove the tag 'sunset'" or "Add tag 'vacation'".

5. Chatbox User Interaction Flow

Make the chat interface truly interactive for this use-case: - **Uploading via Chat:** The user might drag a file into the chat or click the upload button. The UI can then show a chat bubble saying "**User:** [uploaded *filename*] – Description: <*user's description*>". The app (as "System" or "Assistant") can reply in the chat with the outcome: "Stored file *filename.ext* with hash **Xyz123**. Use this hash to retrieve the file." This conversational feedback keeps the user in the loop. - **Requesting a File:** If the user types a hash like `Xyz123` and hits enter, the front-end should recognize that pattern (or attempt to fetch metadata for it). The chat then can display either the file or the gallery as described. For example: - If `Xyz123` is a single file, the chat could show a message: "*Here is file Xyz123:*" followed by either a download link or an inline preview. If previewable (image/pdf), show it directly in the chat. - If `Xyz123` is a parent collection, the response might be: "*Gallery for collection Xyz123:*" and then embed the gallery component. - **Linking Files via Chat:** Possibly allow commands like `link <childHash> to <parentHash>` in the chat. However, it might be easier to manage linking in the UI when uploading (as mentioned, providing the parent hash at upload time). Still, having a command interface is in the spirit of a chat UI. The system could parse

something like “parent ABC123 for XYZ789” to set XYZ’s parent to ABC’s hash. - **OpenAI Canvas-Like Interaction:** We can extend the chat beyond simple requests. For instance, a user could ask in plain English, “Show me all files I uploaded this week” or “Find images of dogs in collection ABC123”. If you integrate an LLM (OpenAI or others), the assistant could interpret such queries, use your API under the hood to filter or search by tags/date, and then return the results. This *conversational command* approach would make the UI very powerful and user-friendly, albeit requiring more advanced NLP integration. It’s an optional stretch goal inspired by Canvas/ChatGPT capabilities.

6. Gallery View and Sorting Features

Implement the gallery page/component to be both visually appealing and functional: - **Layout:** Use a responsive grid layout. You may employ a library or custom CSS for Masonry (especially if thumbnails have varying aspect ratios). Ensure it looks good on both desktop and mobile. - **Preview Generation:** For images/videos, generating thumbnails on upload (server-side) can speed up gallery loading. For example, create a smaller JPEG thumbnail for an image and store it (perhaps in B2 as well) referenced in metadata. Videos could get a screenshot or a low-res preview clip. Documents might use a PDF-to-image for first page, etc. These previews are what the gallery initially loads; the user can click to view the full file. - **Sorting UI:** At the top of the gallery, include a sort dropdown (by Date, Name, Type, Size, Tag). *State-of-the-art sorting* might also include an AI-driven option like “Smart Sort” – this could group similar items (perhaps using tag frequency or visual similarity for images). For example, you could leverage an image embedding model (like OpenAI CLIP) to compute vectors for images and sort by cluster; however, this is an advanced feature. At minimum, provide intuitive sorting and filtering by metadata. - **Filtering/Search:** If a collection has many files, a search bar that filters by file name or tags is useful. This could also be implemented as chips for each tag – clicking a tag filters to files that have that tag. - **Interactive Elements:** Each file in the gallery might have buttons or hover actions: e.g. download, view full size, or even “open in chat”. An “open in chat” could inject that file (or its hash) back into the chat interface, allowing the user to discuss it (if an AI assistant is present for analysis) or get a shareable link, etc. - **Permissions & Sharing:** Although not explicitly requested, you might consider how others access these hashes. If truly anonymous, anyone with the hash can access the file/collection. You might want to generate a *URL* for the gallery view like `yourapp.com/gallery/<hash>` which people can bookmark or share, which would load a page showing that gallery (with the same UI components). Similarly, a direct file link might be `yourapp.com/file/<hash>` for just the raw file (or a minimal page with a preview/download).

Technical Considerations

- **Scalability:** Using direct-to-B2 uploads means your server mainly handles metadata and small control requests, not the file data itself, which is good for scaling. B2 can handle the storage scaling. Ensure your database can handle a large number of records if you expect a massive dataset (millions of files) – a SQL or NoSQL solution should be fine; just index the hash and parentHash fields for quick lookup.
- **Data Integrity:** When using B2, every upload (especially multi-part) should be verified. B2’s API already verifies each part by SHA1 [12]. Storing a checksum in the DB can allow you to double-check integrity or detect corruption.
- **Security:** Since files are accessible by hash, treat the hash like a secret (especially if the data might be sensitive). The hash should be non-guessable (a long random string or hash). You could implement an *expiration* or deletion system if needed (not mentioned, but for completeness – perhaps not needed since it’s for AI dataset accumulation).

- **AI Integration:** Auto-tagging each file has a compute cost. You might use serverless functions or background jobs to handle it so that the upload flow isn't delayed. For instance, once a file is stored, queue a job to do tagging and update the DB later. The chat UI can later show updated tags or allow searching them. If you integrate an LLM for advanced chat commands or content queries (like "summarize this PDF"), ensure you have the necessary AI service and consider rate limits or costs.
- **UX & Polish:** Keep the chat interface responsive. Possibly give the user tooltips or a help message on what to do (e.g., a greeting: "Welcome! Upload a file or enter a hash to begin."). In the gallery, provide a clear indication of the parent collection hash so the user knows what link to share for that gallery.

By combining these components, you'll create an **"omnipotent uploader"** web application: Users can easily upload any file (images, videos, documents, etc.), get a unique hash reference, and organize files by simply grouping under a chosen hash. The chat-style UI makes interactions intuitive, and the integration of Backblaze B2 ensures efficient, scalable storage of potentially massive data (ideal for assembling large AI training datasets). Auto-tagging and an intelligent gallery view will further help users manage and explore the growing repository of content.

Sources:

- Backblaze B2 official sample for direct browser uploads [3] [4] (demonstrates getting upload URLs or presigned URLs for direct client upload; avoids routing files through your server).
- Backblaze B2 documentation on large file uploads (multipart support for files over 5 GB) [6]. Note: single uploads max out at 5 GB, so multipart is used for bigger files.
- Concept of content-addressable storage (as used in IPFS) for generating unique file hashes [1].
- Explanation of AI auto-tagging to automatically label media files with descriptive metadata [2]. This can be achieved via cloud AI services (e.g. Azure Cognitive Services or Google Vision API).

---

[1] How to Share File in IPFS Blockchain? - GeeksforGeeks
https://www.geeksforgeeks.org/ethical-hacking/how-to-share-file-in-ipfs-blockchain/

[2] [10] [11] How to leverage AI for Digital Asset Management with Auto-tagging
https://www.fotoware.com/blog/ai-digital-asset-management-auto-tagging

[3] [4] [5] [6] [9] GitHub - backblaze-b2-samples/b2-browser-upload: Demonstrates uploading files from JavaScript in the browser to Backblaze B2 using both the B2 Native and S3-Compatible APIs
https://github.com/backblaze-b2-samples/b2-browser-upload

[7] [8] [12] How to upload large files in the BackBlaze B2 bucket | by Payoda Technology Inc | Medium
https://payodatechnologyinc.medium.com/how-to-upload-large-files-in-the-backblaze-b2-bucket-5b6bc87bbb5c