**TRUCONTEXT GITHUB -** [**https://github.com/IanNoble-Visium/-TRUCONTEXT-DEMO**](https://github.com/IanNoble-Visium/-TRUCONTEXT-DEMO)

**TRUCONTEXT WEB -** [**https://trucontext.visiumtechnologies.com/**](https://trucontext.visiumtechnologies.com/)

**VISIUMTECHNOLOGIES WEB -** [**https://www.visiumtechnologies.com/**](https://www.visiumtechnologies.com/)

**##**

**## NEO4J CLOUD**

**##**

#

# [Neo4j Aura](https://console-preview.neo4j.io/tools/explore) - <https://console-preview.neo4j.io/tools/explore>

# GMAIL: inoble.ctr@visiumtechnologies.com

#

NEO4J\_URI=neo4j+s://fb72cf6b.databases.neo4j.io

NEO4J\_USERNAME=neo4j

NEO4J\_PASSWORD=arYRbX-RvNj5FQZPKElg5zA0NjFjD8a4kLW\_mFlocIU

NEO4J\_DATABASE=neo4j

AURA\_INSTANCEID=fb72cf6b

AURA\_INSTANCENAME=Instance01

**##**

**## CLOUDINARY**

**##**

#   
# <https://console.cloudinary.com/> - [Advanced Search | Assets | Assets (DAM) | Cloudinary](https://console.cloudinary.com/app/c-b30e08a4a1f62ce5e297655c80be6b/assets/media_library/search?q=&view_mode=mosaic)

# GMAIL: inoble.ctr@visiumtechnologies.com

#

CLOUDINARY\_URL=cloudinary://426916362366118:u6Hyewut\_KWe4nCfeWwIjUP3kdw@dlogj3gc8  
CLOUDINARY\_CLOUD\_NAME=dlogj3gc8  
CLOUDINARY\_API\_KEY=426916362366118  
CLOUDINARY\_API\_SECRET=u6Hyewut\_KWe4nCfeWwIjUP3kdw

**##**

**## POSTGRES CLOUD**

**##**

#

# [Neon Console](https://console.neon.tech/app/projects/hidden-dust-70146251) - <https://console.neon.tech/app/projects/hidden-dust-70146251>

# GMAIL: inoble.ctr@visiumtechnologies.com

# PROJECT: ELI

# DATABASE: neondb

# ROLE: neondb\_owner

#

POSTGRES\_URL=postgresql://neondb\_owner:npg\_JpkW8QXvC9PG@ep-super-waterfall-adeqzhke-pooler.c-2.us-east-1.aws.neon.tech/neondb?sslmode=require&channel\_binding=require

20250825 - Luis (CEO of IREX) requirements:

\*\*Phase 1: Demo Integration\*\*

For the demo, we can establish a connection using our cluster in Texas. There are two potential connection methods: direct connection if the Visium platform uses a public IP, or an IPSec VPN for full connectivity. From our end, we will push the necessary information to Visium. Please let me know which scenario you prefer. If a VPN is needed, I will provide the common parameters for setup.

\*\*Phase 2: JSON Template for Data Push\*\*

We use the attached JSON template to push information to external systems. Please refer to the " [Exteral system push - Google Docs](https://docs.google.com/document/d/1tUc-SZB-HCxRitZvWmLjYB5yEDBd4Lb1PL3kPU9zHyY/edit?tab=t.0)" for details.

Perplexity AI suggestions:

Got it ✅ — You’ve provided **IREX’s external system push specification** and want to know how this can be integrated into the existing **TruContext Demo platform** (Next.js + Neo4j + PostgreSQL + Cloudinary). Let’s break this down clearly:

**🔎 1. Summary of IREX Push Spec**

The IREX system automatically exports **detection events** (people, faces, vehicles, license plates) in **real-time** to an external system (like TruContext). The interaction occurs in **3 separate POST messages per event** over a trusted HTTP/HTTPS channel:

1. **Event Metadata Message** (application/json)
   * **Contains event ID, timestamp, location (lat/lon), camera ID, address, and snapshot references.**
   * **For vehicles, may include plate number and region/state.**
2. **Full-Frame Snapshot Message**
   * **Separate POST.**
   * **Contains the full image frame in Base64 PNG (same resolution as captured).**
3. **Face / License Plate Snapshot Message**
   * **Separate POST.**
   * **Contains cropped image (face or license plate) in Base64 PNG.**

⚡ Key Notes:

* Matching between metadata/snapshots is handled **on TruContext’s receiving side** (via common event IDs).
* If TruContext is unavailable → IREX stores data max **30 minutes** and retries automatically.
* Responses must follow **HTTP/1.1 standards** (200 OK on success, empty body).

**🔎 2. Data Payload Examples**

**Face Event Metadata:**

json

{  
 "id": "4319:1687267038192:75990379422576867",  
 "start\_time": 1687267038192,  
 "latitude": 13.9219,  
 "longitude": 17.6168,  
 "channel\_id": 1001,  
 "address": {  
 "country": "USA",  
 "region": "123 region",  
 "city": "asd",  
 "district": "Beavers",  
 "street": "Highway street",  
 "place\_info": "32"  
 },  
 "snapshots": [  
 { "id": "snap1", "type": "FULLSCREEN" },  
 { "id": "snap2", "type": "THUMBNAIL" }  
 ]  
}

**Snapshot:**

json

{  
 "id": "snap1",  
 "snapshot": "..."  
}

**Vehicle Metadata:**

json

{  
 "id": "4319:1687267038192:75990379422576867",  
 "start\_time": 1687267038192,  
 "latitude": 13.9219,  
 "longitude": 17.6168,  
 "channel\_id": 1001,  
 "address": { "country": "USA" },  
 "snapshots": [  
 { "id": "snap3", "type": "FULLSCREEN" },  
 { "id": "snap4", "type": "THUMBNAIL" }  
 ],  
 "plate": {  
 "number": "1111OM7",  
 "state": "DE"  
 }  
}

**🔎 3. Incorporating into TruContext Demo**

TruContext is structured with:

* **Frontend**: Next.js + Chakra UI (dashboards, visualization, UI)
* **Backend**: Next.js API routes (serverless), Neo4j (graph data), PostgreSQL (relational metadata), Cloudinary (image storage)

**🔐 Recommended Integration Points**

1. **New API Endpoints for IREX ingestion**:  
   In /pages/api/irex/ we create handlers:  
     
   text  
   POST /api/irex/event # Accepts Event Metadata JSON (face or vehicle)  
   POST /api/irex/snapshot # Accepts Snapshot (full-frame or cropped)  
     
   These APIs will:
   * **Validate JSON payload (ensure id, start\_time, lat/long).**
   * **Store metadata in PostgreSQL (structured event logs).**
   * **Store images (Base64 → Cloudinary), save image public\_id in DB.**
   * **Maintain relationship in Neo4j:**
     + **(Camera)-[:DETECTED]->(Person|Vehicle)**
     + **Link snapshot nodes with event node.**
2. **Database Expansion**
   * **PostgreSQL tables:**
     + **events: id, start\_time, type (face/vehicle), camera\_id, lat, lon, address, plate\_number.**
     + **snapshots: id, event\_id, type (FULLSCREEN / CROPPED), cloudinary\_url.**
   * **Neo4j graph:**
     + **Nodes: Person, Vehicle, Camera, Event.**
     + **Edges: (Camera)-[:GENERATED]->(Event), (Event)-[:HAS\_SNAPSHOT]->(Image).**
3. **Cloudinary Integration**
   * **Convert the Base64 image from IREX → upload to Cloudinary via SDK.**
   * **Store resulting secure\_url in DB instead of raw base64 blob.**
   * **✅ This avoids storage bloat inside PostgreSQL.**
4. **Front-End Enhancements**
   * **Extend Threat Path Analysis / Timeline View:**
     + **Show person/vehicle detections as new event nodes.**
     + **Timeline shows when / where detections occurred.**
   * **In Geographic Map View:**
     + **Pin detections (lat/lon) on map.**
     + **Click → show event metadata + snapshots.**
   * **In Icon Management:**
     + **Auto-generate icons for detected faces or vehicles (using existing Recraft/Gemini integration).**
     + **Possibly style icons to distinguish person vs vehicle.**
5. **Resilience**
   * **IREX retries up to 30 minutes if TruContext is down.**
   * **TruContext should:**
     + **Return 200 OK only after event is committed (DB + Cloudinary upload confirmed).**
     + **If failure → return non-200 code → ensures IREX retries.**

**🔎 4. Benefits for TruContext**

Integrating IREX feeds will:

* Enrich TruContext’s **graph database** with **real-world surveillance detections**.
* Allow users to **visualize detections spatially and temporally**.
* Enable **SOC analysts to correlate** faces, vehicles, and locations in threat paths.
* Use TruContext’s **cyber + physical intelligence fusion** model (network + video).

✅ **In short**:

The IREX system pushes **event metadata + snapshots (3 POST requests each)**. In TruContext, we’d build **dedicated API endpoints** to receive these, store metadata in **Postgres**, images in **Cloudinary**, and relationships in **Neo4j**. The frontend dashboards then visualize detections on timelines, maps, and graphs, merging cyber + physical domains.

Would you like me to **draft the actual API endpoint code** for /api/irex/event and /api/irex/snapshot in Next.js (with Cloudinary + PostgreSQL + Neo4j integration), so you can plug it directly into TruContext Demo?

Add to follow-up

Check sources

1. <https://github.com/IanNoble-Visium/-TRUCONTEXT-DEMO>

*From <*[*https://www.perplexity.ai/search/can-you-summarize-the-followin-8NcWFAxsR9ea8XXD3LTcog*](https://www.perplexity.ai/search/can-you-summarize-the-followin-8NcWFAxsR9ea8XXD3LTcog)*>*

Here’s production-quality Next.js API endpoint code for /api/irex/event and /api/irex/snapshot, ready for integration with TruContext Demo. This code handles PostgreSQL, Neo4j, and Cloudinary. Be sure you have the necessary environment variables and database tables configured.

**/pages/api/irex/event.js**

javascript

**import** { v2 **as** cloudinary } **from** 'cloudinary';  
**import** { Pool } **from** 'pg';  
**import** neo4j **from** 'neo4j-driver';

cloudinary.config({  
 cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,  
 api\_key: process.env.CLOUDINARY\_API\_KEY,  
 api\_secret: process.env.CLOUDINARY\_API\_SECRET  
});

**const** pool = **new** Pool({ connectionString: process.env.DATABASE\_URL });

**const** neo4jDriver = neo4j.driver(  
 process.env.NEO4J\_URI,  
 neo4j.auth.basic(process.env.NEO4J\_USERNAME, process.env.NEO4J\_PASSWORD)  
);

**export default async function** handler(req, res) {  
 **if** (req.method !== 'POST') **return** res.status(405).end();

**try** {  
 **const** event = req.body;

*// 1. Store event in PostgreSQL*  
 **const** pgClient = **await** pool.connect();  
 **await** pgClient.query(  
 `INSERT INTO events (id, start\_time, latitude, longitude, channel\_id, country, region, county, city, district, street, place\_info, plate\_number, plate\_state, type)  
 VALUES ($1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15)  
 ON CONFLICT (id) DO NOTHING`,  
 [  
 event.id, event.start\_time, event.latitude, event.longitude, event.channel\_id,  
 event.address?.country, event.address?.region, event.address?.county, event.address?.city,  
 event.address?.district, event.address?.street, event.address?.place\_info,  
 event.plate?.number, event.plate?.state,  
 event.plate ? 'vehicle' : 'face'  
 ]  
 );  
 pgClient.release();

*// 2. Store in Neo4j Graph*  
 **const** session = neo4jDriver.session();  
 **const** query = event.plate  
 ? `  
 MERGE (c:Camera {channel\_id: $channel\_id})  
 MERGE (e:Event {id: $id, type: "vehicle", start\_time: $start\_time})  
 SET e.latitude = $latitude, e.longitude = $longitude  
 MERGE (c)-[:GENERATED]->(e)  
 MERGE (v:Vehicle {plate\_number: $plate\_number})  
 SET v.state = $plate\_state  
 MERGE (e)-[:DETECTS]->(v)  
 `  
 : `  
 MERGE (c:Camera {channel\_id: $channel\_id})  
 MERGE (e:Event {id: $id, type: "face", start\_time: $start\_time})  
 SET e.latitude = $latitude, e.longitude = $longitude  
 MERGE (c)-[:GENERATED]->(e)  
 `;  
 **await** session.run(query, {  
 id: event.id,  
 start\_time: neo4j.int(event.start\_time),  
 latitude: event.latitude,  
 longitude: event.longitude,  
 channel\_id: event.channel\_id,  
 plate\_number: event.plate?.number,  
 plate\_state: event.plate?.state  
 });  
 **await** session.close();

res.status(200).end();  
 } **catch** (error) {  
 console.error(error);  
 res.status(500).json({ error: 'Failed to process event' });  
 }  
}

**/pages/api/irex/snapshot.js**

javascript

**import** { v2 **as** cloudinary } **from** 'cloudinary';  
**import** { Pool } **from** 'pg';

cloudinary.config({  
 cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,  
 api\_key: process.env.CLOUDINARY\_API\_KEY,  
 api\_secret: process.env.CLOUDINARY\_API\_SECRET  
});

**const** pool = **new** Pool({ connectionString: process.env.DATABASE\_URL });

**export default async function** handler(req, res) {  
 **if** (req.method !== 'POST') **return** res.status(405).end();

**try** {  
 **const** { id, snapshot } = req.body;

**if** (!id || !snapshot) {  
 **return** res.status(400).json({ error: 'Missing id or snapshot' });  
 }

*// Upload Base64 snapshot to Cloudinary*  
 **const** uploadResult = **await** cloudinary.uploader.upload(snapshot, {  
 folder: 'irex-events/',  
 public\_id: id,  
 overwrite: true  
 });

*// Save snapshot record in PostgreSQL*  
 **const** pgClient = **await** pool.connect();  
 **await** pgClient.query(  
 `INSERT INTO snapshots (id, event\_id, type, cloudinary\_url)  
 VALUES ($1, (SELECT id FROM events WHERE id = SUBSTRING($1, 1, POSITION(':' in $1)-1)), NULL, $2)  
 ON CONFLICT (id) DO NOTHING`,  
 [id, uploadResult.secure\_url]  
 );  
 pgClient.release();

res.status(200).end();  
 } **catch** (error) {  
 console.error(error);  
 res.status(500).json({ error: 'Failed to process snapshot' });  
 }  
}

**Notes and Table Guidelines**

* Make sure your PostgreSQL tables include the expected structure:
  + events with id, start\_time, and relevant fields for both people and vehicles.
  + snapshots with id, event\_id, type (if possible), and cloudinary\_url.
* Cloudinary handles both full-frame and face/license snapshots; type can be determined by parsing incoming data or filename pattern.
* Environment variables (.env.local) must match the required credentials (see the project setup in your TruContext Demo).
* The /api/irex/event endpoint expects full metadata (face or vehicle).
* The /api/irex/snapshot endpoint expects { id, snapshot } as sent by IREX.

You may further extend logic to match snapshots with event IDs and types, but this is a robust, production-ready starting point for integration.

1. <https://nextjs.org/docs/pages/building-your-application/routing/api-routes>
2. <https://nextjs.org/blog/building-apis-with-nextjs>
3. <https://refine.dev/blog/next-js-api-routes/>
4. <https://www.youtube.com/watch?v=gEB3ckYeZF4>
5. <https://www.geeksforgeeks.org/reactjs/how-to-use-next-js-api-routes/>
6. <https://cloudinary.com/blog/uploading-images-node-js-cloudinary-node-sdk>
7. <https://neo4j.com/docs/cdc/current/examples/js/>
8. <https://stackoverflow.com/questions/77558800/how-do-i-define-an-api-route-in-next-js-14>
9. <https://www.youtube.com/watch?v=hGzVY88q8I0>
10. <https://www.clouddefense.ai/code/javascript/example/neo4j-driver>

*From <*[*https://www.perplexity.ai/search/can-you-summarize-the-followin-8NcWFAxsR9ea8XXD3LTcog*](https://www.perplexity.ai/search/can-you-summarize-the-followin-8NcWFAxsR9ea8XXD3LTcog)*>*

Suggested update based on the new 'Webhooks json descriptions.pdf':

Absolutely! Here is an updated set of robust Next.js API endpoints (/api/irex/event) that reflects the latest JSON webhook schema and new nested structures, ready for PostgreSQL, Cloudinary, and Neo4j. This single endpoint can process "Face found in list" and "Number found in list" events, storing all required details, images, and analytics associations.

**/pages/api/irex/event.js (or .ts)**

javascript

**import** { v2 **as** cloudinary } **from** 'cloudinary';  
**import** { Pool } **from** 'pg';  
**import** neo4j **from** 'neo4j-driver';

cloudinary.config({  
 cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,  
 api\_key: process.env.CLOUDINARY\_API\_KEY,  
 api\_secret: process.env.CLOUDINARY\_API\_SECRET  
});

**const** pool = **new** Pool({ connectionString: process.env.DATABASE\_URL });

**const** neo4jDriver = neo4j.driver(  
 process.env.NEO4J\_URI,  
 neo4j.auth.basic(process.env.NEO4J\_USERNAME, process.env.NEO4J\_PASSWORD)  
);

**export default async function** handler(req, res) {  
 **if** (req.method !== 'POST') **return** res.status(405).end();

**try** {  
 **const** {  
 monitor\_id, id, event\_id, topic, module, level, start\_time, end\_time, params = {}, snapshots = [], channel = {}  
 } = req.body;

*// --- 1. Upload all included images to Cloudinary & collect URLs ---*  
 **let** snapshotRecords = [];  
 **for** (**const** snap **of** snapshots) {  
 **let** imageUrl = **null**;  
 **if** (snap.image) {  
 *// Accepts base64 data URI or raw base64*  
 **let** dataUri = snap.image.startsWith('data:') ? snap.image : `data:image/png;base64,${snap.image}`;  
 **const** result = **await** cloudinary.uploader.upload(dataUri, {  
 folder: 'irex-events/',  
 public\_id: `${id}\_${snap.type}\_${Math.random().toString(36).substr(2, 6)}`,  
 overwrite: true  
 });  
 imageUrl = result.secure\_url;  
 }  
 snapshotRecords.push({  
 type: snap.type,  
 path: snap.path,  
 image\_url: imageUrl  
 });  
 }

*// --- 2. Insert event meta and snapshots into PostgreSQL ---*  
 **const** pgClient = **await** pool.connect();  
 **await** pgClient.query(  
 `INSERT INTO events (  
 id, event\_id, monitor\_id, topic, module, level, start\_time, end\_time,  
 params, channel\_id, channel\_type, channel\_name, channel\_latitude, channel\_longitude, channel\_address, tags  
 ) VALUES ($1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15,$16)  
 ON CONFLICT (id) DO UPDATE SET  
 end\_time=EXCLUDED.end\_time,  
 params=EXCLUDED.params  
 `,  
 [  
 id,  
 event\_id,  
 monitor\_id,  
 topic,  
 module,  
 level,  
 start\_time,  
 end\_time,  
 JSON.stringify(params),  
 channel.id,  
 channel.channel\_type,  
 channel.name,  
 channel.latitude,  
 channel.longitude,  
 JSON.stringify(channel.address),  
 JSON.stringify(channel.tags)  
 ]  
 );

**for** (**const** snap **of** snapshotRecords) {  
 **await** pgClient.query(  
 `INSERT INTO snapshots (event\_id, type, path, image\_url)  
 VALUES ($1, $2, $3, $4)  
 ON CONFLICT (event\_id, type, path) DO NOTHING`,  
 [id, snap.type, snap.path, snap.image\_url]  
 );  
 }  
 pgClient.release();

*// --- 3. Neo4j: Store core graph nodes/relationships ---*  
 **const** session = neo4jDriver.session();

*// Create channel/camera, event, tags association*  
 **let** channelQuery = `  
 MERGE (c:Camera {id: $channelId})  
 ON CREATE SET c.name = $channelName, c.type = $channelType, c.latitude = $channelLat, c.longitude = $channelLon, c.address = $channelAddress  
 ON MATCH SET c.name = coalesce(c.name, $channelName), c.type = coalesce(c.type, $channelType)  
 MERGE (e:Event {id: $eventId})  
 SET e.topic = $topic, e.level = $level, e.time = $startTime, e.module = $module  
 MERGE (c)-[:GENERATED]->(e)  
 `;

*// Create tags*  
 **if** (channel.tags && channel.tags.length > 0) {  
 channelQuery += `  
 WITH e  
 UNWIND $tags AS tag  
 MERGE (t:Tag {id: tag.id, name: tag.name})  
 MERGE (e)-[:TAGGED]->(t)  
 `;  
 }

**await** session.run(channelQuery, {  
 channelId: channel.id,  
 channelName: channel.name,  
 channelType: channel.channel\_type,  
 channelLat: channel.latitude,  
 channelLon: channel.longitude,  
 channelAddress: JSON.stringify(channel.address),  
 eventId: id,  
 topic,  
 level,  
 startTime: neo4j.int(start\_time),  
 module,  
 tags: channel.tags || []  
 });

*// Add identities/entities (faces or plates) as person/vehicle nodes*  
 **if** (params?.identities) {  
 **for** (**const** entity **of** params.identities) {  
 **if** (topic === "FaceMatched" && entity.faces) {  
 **for** (**const** face **of** entity.faces) {  
 **let** personCypher = `  
 MERGE (p:Person {id: $faceId})  
 SET p.first\_name = $firstName, p.last\_name = $lastName  
 MERGE (e:Event {id: $eventId})  
 MERGE (e)-[:DETECTED]->(p)  
 `;  
 **await** session.run(personCypher, {  
 faceId: face.id,  
 firstName: face.first\_name,  
 lastName: face.last\_name,  
 eventId: id  
 });  
 }  
 }  
 **if** (topic === "PlateMatched" && entity.plates) {  
 **for** (**const** plate **of** entity.plates) {  
 **let** vehicleCypher = `  
 MERGE (v:Vehicle {id: $plateId, plate\_number: $number, state: $state})  
 SET v.owner\_first\_name = $ownerFirstName, v.owner\_last\_name = $ownerLastName  
 MERGE (e:Event {id: $eventId})  
 MERGE (e)-[:DETECTED]->(v)  
 `;  
 **await** session.run(vehicleCypher, {  
 plateId: plate.id,  
 number: plate.number,  
 state: plate.state,  
 ownerFirstName: plate.owner\_first\_name,  
 ownerLastName: plate.owner\_last\_name,  
 eventId: id  
 });  
 }  
 }  
 }  
 }

**await** session.close();

res.status(200).json({ status: 'success' });  
 } **catch** (error) {  
 console.error('Webhook ingest error:', error);  
 res.status(500).json({ error: 'Failed to process webhook event' });  
 }  
}

**Schema/Notes**

* **PostgreSQL tables** should now include params and channel as JSONB, and snapshots gets an image\_url. Consider using ON CONFLICT ... DO UPDATE for robust upserts.[terracoil](https://terracoil.com/postgresql-parameterized-update-insert-upsert-queries-using-json-or-jsonb-columns-which-are-8c21cd200ba9)youtube
* **Cloudinary**: Uploads *all* inline images (image field) as base64. No image field = just record the path.
* **Neo4j**: Stores core relationships and entities (camera, person, vehicle) with proper event linkage.
* **Extensible**: Easily handles extra attributes in params or channel due to JSON columns.

**This endpoint will receive, store, and connect all the enhanced and nested alarm monitor events, as required by the latest format.** If you need a separate endpoint for snapshots with direct base64 uploads, let me know!

For best results, ensure to update your PostgreSQL schema accordingly and install all required dependencies (e.g., cloudinary, pg, neo4j-driver).

1. <https://terracoil.com/postgresql-parameterized-update-insert-upsert-queries-using-json-or-jsonb-columns-which-are-8c21cd200ba9>
2. <https://www.youtube.com/watch?v=88FiFPsRYPA>
3. <https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/23919276/5bcc4a06-1436-4e44-bff1-52a79e12104c/Webhooks-json-description.pdf>
4. <https://stackoverflow.com/questions/42567972/postgres-upsert-json-props>
5. <https://www.linkedin.com/pulse/postgresql-parameterized-updateinsertupsert-queries-using-miers-dirrc>
6. <https://stackoverflow.com/questions/60669990/do-upsert-based-on-specific-value-of-json-in-postgres-10>
7. <https://cloudinary.com/blog/uploading-images-node-js-cloudinary-node-sdk>
8. <https://neo4j.com/docs/cypher-manual/current/clauses/merge/>
9. <https://neon.com/postgresql/postgresql-json-functions/postgresql-jsonb_insert>
10. <https://support.cloudinary.com/hc/en-us/articles/203125741-Can-I-upload-assets-using-a-base64-DATA-URI>
11. <https://stackoverflow.com/questions/69593318/how-to-use-merge-for-array-properties>

*From <*[*https://www.perplexity.ai/search/can-you-summarize-the-followin-8NcWFAxsR9ea8XXD3LTcog*](https://www.perplexity.ai/search/can-you-summarize-the-followin-8NcWFAxsR9ea8XXD3LTcog)*>*