CROSS-CLOUD EVENT-DRIVEN STORAGE REPLICATOR(AWS-GCS)

1. **Project Overview**

This project is an event-driven cross-cloud file replication service that automatically copies files from AWS S3 to Google Cloud Storage (GCS) the moment they are uploaded to S3.

The system is built around the following core components:

* FastAPI service – provides a /v1/replicate HTTP POST endpoint to handle replication requests.
* AWS Lambda trigger – listens to S3 object creation events and sends replication requests to the FastAPI endpoint.
* GCS integration – securely uploads files into a target Google Cloud Storage bucket using a service account key.

**How it works**:

1. A file is uploaded to the configured AWS S3 bucket.
2. S3 emits an event notification, which invokes an AWS Lambda function.
3. The Lambda function extracts the object details (bucket, key) and sends them to the FastAPI /v1/replicate endpoint.
4. The FastAPI service downloads the file from S3 and streams it to the target GCS bucket.

This architecture ensures real-time replication, idempotency (safe re-runs without duplicate uploads), and secure credential handling via environment variables.

**Tech Stack**

* AWS S3 – Source storage
* AWS Lambda – Event trigger
* FastAPI – Replication service API
* boto3 – AWS SDK for Python
* google-cloud-storage – GCP SDK for Python
* Render – Hosting the FastAPI service
* Python 3.9+

1. **Create Python Virtual Environment**

python -m venv venv

source venv/bin/activate # Linux/Mac

venv\Scripts\activate # Windows

1. **Install Requirements**

pip install -r requirements.txt

fastapi

uvicorn

boto3

google-cloud-storage

python-dotenv

1. **How AWS ↔ GCS Integration Works**

When a new object is created in an S3 bucket, S3 sends an event that invokes an AWS Lambda. The Lambda posts the object details to the FastAPI /v1/replicate endpoint. FastAPI streams the object from S3 to GCS, checking idempotency and retrying transient failures.

**4.1 — Prerequisites (what must exist)**

* AWS account with an S3 bucket (source).
* AWS Lambda (to forward S3 events).
* GCP project with a GCS bucket (destination).
* GCP service account JSON with Storage Object Admin on the target bucket.
* FastAPI service hosted and reachable by Lambda (Render, ngrok, etc.).
* Local requirements.txt includes fastapi, boto3, google-cloud-storage, uvicorn.

**4.2 — S3 event notification (how to set it)**

1. Open S3 Console → Buckets → [your-bucket] → Properties → Event notifications.
2. Create a new notification:
   * Name: TriggerReplicationOnUpload (or your choice).
   * Event types: All object create events (or PUT).
   * Destination: Lambda function → choose your Lambda.
   * Leave Prefix/Suffix blank (or use filters if you only want certain keys).
3. Save.

**4.3 — Lambda function**

Purpose: quickly forward S3 events to the external FastAPI endpoint. This keeps Lambda small and avoids bundling heavy GCP libs in Lambda.

Required Lambda environment variable

* FASTAPI\_URL — e.g. https://cross-cloud-storage-replicator.onrender.com

Minimal Lambda handler (copy into lambda\_function.py):

# lambda\_function.py

import os, json, urllib.parse

import urllib3

http = urllib3.PoolManager()

FASTAPI\_URL = os.environ.get("FASTAPI\_URL") # must be set in Lambda env

def lambda\_handler(event, context):

# pick the first record

rec = event["Records"][0]["s3"]

bucket = rec["bucket"]["name"]

raw\_key = rec["object"]["key"]

key = urllib.parse.unquote\_plus(raw\_key) # decode + and %20

payload = {"bucket": bucket, "key": key}

resp = http.request(

"POST",

f"{FASTAPI\_URL}/v1/replicate",

body=json.dumps(payload),

headers={"Content-Type":"application/json"},

timeout=30.0

)

print("FastAPI status:", resp.status, resp.data)

return {"statusCode": 200}

Notes:

* Full code is stored in lambda\_function.py. This is just sample.
* Lambda does not need S3 read permissions in this model (FastAPI does the S3 read). If you instead implement replication in Lambda, grant s3:GetObject and bundle google-cloud-storage.
* Make sure the Lambda function is allowed to be invoked by your S3 bucket (S3 config adds this automatically in most consoles).

**4.4 — FastAPI replication service (what it must do)**

Purpose: receive { "bucket": "...", "key": "..." }, stream the object from S3 and stream upload to GCS.

Essential responsibilities

1. Validate request — ensure bucket and key present.
2. Decode key — S3 encodes spaces/special characters; decode with urllib.parse.unquote\_plus.
3. Idempotency check — if the same object (same name and size) exists in GCS, skip upload.
4. Streaming transfer:
   * Use boto3.get\_object(...)[‘Body’] (StreamingBody) and read in chunks (e.g., 8MB).
   * Use google.cloud.storage blob.open("wb") or blob.upload\_from\_file() with the stream.
5. Retry wrapper — for transient network errors (3 attempts with backoff).
6. Logging — INFO for start/complete, ERROR on exceptions (Render logs show these).

Key code snippets to include in FastAPI:

* Decode S3 key:

import urllib.parse

raw\_key = record["s3"]["object"]["key"]

s3\_key = urllib.parse.unquote\_plus(raw\_key)

* Streaming download from S3:

resp = s3.get\_object(Bucket=bucket, Key=key)

s3\_stream = resp["Body"] # StreamingBody

* Streaming upload to GCS:

blob = gcs\_bucket.blob(dest\_name)

with blob.open("wb") as f:

while True:

chunk = s3\_stream.read(8\*1024\*1024)

if not chunk:

break

f.write(chunk)

* Idempotency check:

blob = gcs\_bucket.blob(dest\_name)

if blob.exists() and blob.size == s3\_size:

# skip upload

Environment variables (FastAPI / Render)

* AWS\_ACCESS\_KEY\_ID
* AWS\_SECRET\_ACCESS\_KEY
* AWS\_REGION (e.g. ap-south-1)
* GCS\_BUCKET or GCP\_BUCKET\_NAME
* GCP\_SA\_KEY\_B64 (base64 of service account JSON) or GCP\_SA\_KEY (raw JSON) — see below for storage suggestions
* Optional: RETRY\_ATTEMPTS, RETRY\_BACKOFF\_SEC

**4.5 — GCP service account setup & permissions**

1. Create a service account in GCP IAM (replicator-service-account).
2. Grant it Storage Object Admin on the target bucket or project.
3. Generate a JSON key and store it securely (Render secret, SSM, or base64 in env var).
4. If using base64 in Lambda env, set GCP\_SA\_KEY\_B64=<base64(service-account.json)>. For Render, upload the JSON file and set GCP\_CREDENTIALS\_PATH or provide content in an env var depending on your hosting method.

**4.6 — Packaging & deployment notes**

* FastAPI (Render): push code + requirements.txt to GitHub; configure Render service and add environment variables in the Render dashboard. Render will install deps.
* Lambda: if Lambda only forwards to FastAPI (above), the deployment is a small Python file — no heavy packaging required.
* If Lambda will do replication itself: you must package google-cloud-storage (and dependencies) with your Lambda (use pip install -t . and zip contents), or use a Lambda Layer / container image. If zip > 50MB, upload to S3 and choose “Upload from S3” in Lambda console.

**4.7 — Testing checklist (exact steps to verify)**

1. Start FastAPI (if testing locally): uvicorn app:app --reload and expose via ngrok OR deploy to Render and note public URL.(I used render here)
2. Set Lambda FASTAPI\_URL to the FastAPI public URL (Render URL).
3. Upload file to S3 (console or CLI)
4. Check AWS CloudWatch (Lambda logs) — confirm Lambda logged event and POST result.
5. Check Render logs — confirm FastAPI received request and logged “Downloaded … Uploaded …”.
6. Open GCS Console → target bucket → confirm file exists.
7. Re-upload same file (same key) — confirm FastAPI/Lambda logs show “skipped — already exists”.

**4.8 — Common errors & how to fix them**

* NoSuchKey when calling get\_object  
  Cause: key contains encoded spaces (+ or %20).  
  Fix: urllib.parse.unquote\_plus(key) before calling S3.
* Runtime.ImportModuleError: No module named 'google'  
  Cause: google-cloud-storage not included in Lambda package.  
  Fix: Either forward to FastAPI (Lambda stays lightweight) or package google-cloud-storage into Lambda or use a Layer.
* Request must be smaller than 5120 bytes updating environment  
  Cause: you pasted large JSON into Lambda environment variable.  
  Fix: store large secret in AWS SSM Parameter Store and reference it (${ssm:/myapp/gcp-service-account}) or use base64 in SSM.
* google-crc32c warning  
  Cause: CRC C extension not compiled.  
  Fix: Warning only — works with pure-Python fallback. For performance, compile google-crc32c in build environment.
* PermissionDenied / 403 on GCS upload  
  Cause: GCP service account lacks Storage Object Admin for bucket.  
  Fix: Grant role to the service account for the bucket.
* Lambda not triggered when uploading  
  Cause: S3 event notification misconfigured or upload happened before notification was created.

**4.9 — Security & cleanup checklist**

* Never commit keys or .env to GitHub. Add .env to .gitignore.
* After testing, delete any downloaded GCP JSON keys and revoke old keys via GCP Console.
* Consider using AWS Secrets Manager / SSM or GCP Secret Manager for production credentials.

1. **Idempotency & Error-Handling**

Goal: make each S3 → GCS replication safe to run multiple times for the same S3 object without creating duplicates, partial files, or data corruption; and make transient failures recoverable with retries and clear observability for permanent failures.

**Idempotency strategy (what we do and why)**

**5.1-Fast existence check (size-based):**Before uploading, check whether a blob with the same name exists in GCS and whether its stored size equals the S3 object's ContentLength.If sizes match, assume the object is already replicated and skip the upload. This is fast and avoids re-uploading the same bytes.

**5.2-Atomic upload with a temporary name:**To avoid partial/visible objects if upload fails mid-way, upload to a temporary blob name first (e.g. path/to/file.ext.\_\_tmp\_\_{uuid4()}), then rename (copy) to the final name only after upload completes.Copy + delete is atomic from the user's perspective: the final name appears only after a successful complete upload. This prevents partially-written files being treated as valid.

**5.3-Precondition on final write (race protection)**:When moving the temp blob into the final name, use precondition checks (if available) to avoid overwriting an existing file created by another concurrent worker. GCS supports preconditions like if\_generation\_match=0 for uploads to ensure you don’t accidentally overwrite an existing object.

**5.4-Optional stronger fingerprinting (when you need it):** For absolute certainty, compare checksums (MD5 or CRC32C). Note: S3 ETag is only MD5 for single-part uploads; multipart uploads have ETag that’s not a simple MD5. Computing a checksum requires reading the object (extra cost) or storing checksums at upload time. Choose this only if required by data integrity policy.

**5.5-Why this is good enough for the take-home:**

* Size-checking prevents most duplicate uploads with minimal cost and latency.
* Temporary-name + rename prevents partially uploaded objects from being seen as complete.
* Precondition prevents subtle race conditions if two events arrive at nearly the same time.
* Also, in lambda code there is no resource for fast api process because here there is no use for manual testing as we are not doing in big size but, in future we can add it so error handling can be easy and also the process will have some more added properties.

**6.Features / Requirements Met**

This project fulfills all criteria specified in the assessment:

1. Event-Driven Architecture
   * Uses AWS S3 Event Notifications to automatically trigger AWS Lambda when a new object is uploaded.
   * Fully automated — no manual intervention needed after file upload.
2. Cross-Cloud File Replication
   * Files are replicated from AWS S3 to Google Cloud Storage (GCS).
   * Works across cloud providers seamlessly.
3. Streaming-Based Transfer
   * No local file storage — the object is streamed directly from S3 to GCS, reducing memory/disk usage.
4. Idempotency
   * Checks if the object already exists in GCS before uploading to avoid duplicate uploads.
5. Retry Mechanism
   * Retries transient failures (e.g., network issues, API timeouts) to ensure reliable delivery.
6. Secure Configuration
   * AWS and GCP credentials are stored in environment variables and GCP JSON key files.
   * No sensitive data is hardcoded in the source code.
7. Decoupled Design
   * AWS Lambda is lightweight and only forwards event data to FastAPI.
   * FastAPI handles all replication logic, allowing independent scaling.
8. REST API Endpoint
   * /v1/replicate endpoint implemented using FastAPI.
   * Accepts JSON payload with S3 bucket and key details for manual or automated replication.
9. Logging & Monitoring
   * AWS Lambda logs to CloudWatch for trigger events.
   * FastAPI logs all replication activity for debugging and auditing.
10. Extensibility
    * Architecture allows adding other storage providers in the future (e.g., Azure Blob Storage) with minimal changes.

**7.Conclusion**

This project successfully demonstrates event-driven, cross-cloud file replication from AWS S3 to Google Cloud Storage using a FastAPI service triggered by AWS Lambda. It meets the given requirements for streaming, idempotency, retries, and secure configuration, while leaving room for future enhancements in security, scalability, and monitoring

**License / Author**

Developed by Kunguma Dharshini M as part of a take-home assessment.