AWS Lambda Function for Email2Webhook

AWS Lambda Function for Email2Webhook

Anonymous · 22/09/2026 · #aws, #lambda, #python

AWS Lambda Function for Email2Webhook

github derafu/aws-lambda-email2webhook CI passing

AWS Lambda Function that receives inbound email through Amazon SES and forwards it as a signed HTTP webhook, so a downstream system (for example, a support ticketing workflow) can process it.

How it works

Amazon SES invokes this function directly through a receipt rule's **Lambda action** (no SNS topic involved). The Lambda action never delivers the email body, only metadata, so the receipt rule must also have an **S3 action placed before the Lambda action** to store the raw MIME message. This function:

1. Reads the `messageId` from the SES event.
2. Downloads the raw email from the S3 bucket (object key = `messageId`, optionally prefixed).
3. Builds an envelope: `{"meta": {"source": "aws-ses", "format": "...", "version": "1.0.0"}, "data": {...}}`. `meta.format` (`WEBHOOK_FORMAT`) selects which builder produces `data`, so the shape of `data` is fully determined by `format`. See "Output formats" below.
4. Signs the envelope (`WEBHOOK_SIGNATURE`) and sends it as a POST request to the configured webhook.

Output formats (`WEBHOOK_FORMAT`)

- `postal` (default) — `data` matches the JSON shape Postal sends for HTTP endpoints configured with `Encoding=BodyAsJSON` and `Format=Hash` (verified against Postal's own source, `app/senders/http_sender.rb`), so an existing Postal-based integration can receive it without changes to its parsing logic. Two fields have no real SES equivalent and are approximated:
    - `id` / `token` are Postal's internal sequential integer ID and per-message secret; both are filled with SES's `messageId` (a string, not an integer) since nothing downstream is known to rely on `id` being numeric.
    - `spam_status` is derived from SES's `spamVerdict.status` via the `SPAM_STATUS_MAP` dict in `lambda_function.py` (edit it directly if the mapping needs to change) — `GRAY` and `PROCESSING_FAILED` are mapped to `"Spam"` on purpose: an uncertain or failed scan

    shouldn't be trusted as much as a `PASS`, so it gets the same downstream priority as a confirmed `FAIL` instead of being silently treated as clean.
    - `received_with_ssl` is always `null` — SES doesn't report whether the original SMTP session used TLS.
    - `bounce` is always `false` — an inbound SES message is never a bounce notification.

- `ses` — `data` is the envelope's own native representation, no MIME parsing performed by this function: `{"ses": {"mail": {...}, "receipt": {...}}, "email_base64": "<raw .eml, base64>"}`. Use this for a receiver that wants to parse the MIME message itself.

Adding a new output format means adding one function to `DATA_BUILDERS` in `lambda_function.py` — nothing else in the function needs to change.

Signing (`WEBHOOK_SIGNATURE`)

- `hmac` (default) — HMAC-SHA256 of the exact request body, sent as `X-Webhook-Signature-256: <hex digest>`, with no prefix by default. `WEBHOOK_SIGNATURE_PREFIX` prepends any literal string to the digest if the receiving end expects one (e.g. set it to `sha256=` for the convention GitHub webhooks use) — see "Verifying the webhook signature" below.
- `none` — no signature header sent.

There is intentionally no `rsa` option that mimics Postal's own signature scheme (`X-Postal-Signature-256`, RSA-SHA256 signed with Postal's private key): that private key lives on the Postal server, and reusing it here would mean sharing one signing key across two independent systems, which is the wrong tradeoff for what it saves. If your receiving system already authenticates some other source using its own signature scheme, add a **separate branch or webhook trigger** for this Lambda's `hmac` signature instead of trying to make this function reproduce that other scheme — the `format: "postal"` envelope already gets the `data` shape right, only the authentication step needs its own path.

Required AWS setup

This guide assumes the Lambda function itself already exists (`python3.14` runtime), with the AWS-managed `AWSLambdaBasicExecutionRole` attached to its execution role — that part is generic Lambda setup, not specific to this project. Everything below covers wiring SES → S3 → this Lambda, using the AWS CLI. Replace every `<PLACEHOLDER>` with your own values, and every plain example value (account `123456789012`, region `us-east-2`, names) with your own; the commands below assume `aws configure` is already set up with credentials that can manage S3, SES, IAM and Lambda.

At a high level, you need:

- An S3 bucket that the SES service principal is allowed to write to.

- An SES receipt rule with two actions, in this order:
    1. **S3** — store the raw message in the bucket above.
    2. **Lambda** — invoke this function, invocation type `Event` (recommended; `RequestResponse` also works, the function returns a `disposition` for that case).
- The Lambda execution role needs `s3:GetObject` on that bucket (and prefix, if any).
- The Lambda function's own **Timeout** must be raised well above the runtime's default of 3 seconds — see "Lambda timeout" below.

The rest of this section walks through each of these with the actual CLI commands.

S3 bucket: creation and permissions

Namespace. When creating the bucket, prefer S3's **account regional namespace** over the shared global one (the console offers both, account regional is the one marked "recommended"). It only has to be unique within your own account and region, it can never be re-created by another AWS account after you delete it, and it works exactly like any other general purpose bucket — no limitation on SES writing to it. The console (and the CLI, if you follow the same convention) appends `-<accountId>-<region>-an` to whatever prefix you choose, e.g. an `email-inbox` prefix in account `123456789012` and region `us-east-2` becomes `email-inbox-123456789012-us-east-2-an`.

Three separate permissions are involved, easy to confuse for a single "give access" step:

Who → who	Permission	Where it's configured
SES → S3	`s3:PutObject` (write the raw email)	**bucket policy** on the S3 bucket
SES → Lambda	`lambda:InvokeFunction`	**resource policy** on the Lambda function
Lambda → S3	`s3:GetObject` (read the raw email back)	**execution role** attached to the Lambda function

If you configure the S3 and Lambda actions through the SES console's receipt rule editor, it adds the first two automatically. The steps below use the CLI instead, so all three have to be set up explicitly.

1. Look up your account ID and create the bucket

```
$ aws sts get-caller-identity --query Account --output text
123456789012

$ aws s3api create-bucket \
    --bucket email-inbox-123456789012-us-east-2-an \
    --region us-east-2 \
    --create-bucket-configuration LocationConstraint=us-east-2
```

```
{
    "Location": "http://email-inbox-123456789012-us-east-2-an.s3.amazonaws.com/"
}
```

(`--create-bucket-configuration LocationConstraint=<region>` is required for every region except `us-east-1`, which is the one region `create-bucket` accepts with no location constraint at all — a historical quirk from before S3 supported multiple regions.)

2. Find (or create) the SES receipt rule set you'll use

```
$ aws ses describe-active-receipt-rule-set --query 'Metadata.Name' --output text
default-rule-set
```

If there's no active rule set yet:

```
$ aws ses create-receipt-rule-set --rule-set-name default-rule-set
$ aws ses set-active-receipt-rule-set --rule-set-name default-rule-set
```

3. Attach a bucket policy allowing SES to write into it. Save this as `bucket-policy.json`, replacing `<BUCKET_NAME>`, `<ACCOUNT_ID>`, `<REGION>`, `<RULE_SET_NAME>` and `<RECEIPT_RULE_NAME>` (the receipt rule is created in the next step, but its name must match exactly here):

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowSESPuts",
            "Effect": "Allow",
            "Principal": {
                "Service": "ses.amazonaws.com"
            },
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::<BUCKET_NAME>/*",
            "Condition": {
                "StringEquals": {
                    "AWS:SourceAccount": "<ACCOUNT_ID>",
                    "AWS:SourceArn": "arn:aws:ses:<REGION>:<ACCOUNT_ID>:receipt-rule-
set/<RULE_SET_NAME>:receipt-rule/<RECEIPT_RULE_NAME>"
                }
            }
        }
    ]
}
```

Then apply it (no output on success):

```
$ aws s3api put-bucket-policy \
    --bucket email-inbox-123456789012-us-east-2-an \
    --policy file://bucket-policy.json
```

Notes on this policy:

- `Resource` ends in `/*` — the permission is on objects inside the bucket, not on the bucket itself.
- The `Condition` block locks this down to **this specific receipt rule** (`AWS:SourceArn`) inside **this specific account** (`AWS:SourceAccount`) — without it, any SES account writing through any rule could write to the bucket.
- If the rule set or the rule is ever renamed, `AWS:SourceArn` has to be updated to match, or SES puts will start failing with a "Could not write to bucket" error.
- `SES_S3_KEY_PREFIX` (see "Environment variables" below) is a separate, independent setting — it doesn't appear in this policy at all, it only affects which key inside the bucket the object gets written to.

SES receipt rule and Lambda invoke permission

4. Create the receipt rule, with the S3 action before the Lambda action, replacing the recipient address, bucket name/prefix and Lambda function ARN with your own (no output on success):

```
$ aws ses create-receipt-rule \
    --rule-set-name default-rule-set \
    --rule '{
        "Name": "email2webhook",
        "Enabled": true,
        "TlsPolicy": "Optional",
        "ScanEnabled": true,
        "Recipients": ["inbox@example.com"],
        "Actions": [
            {
                "S3Action": {
                    "BucketName": "email-inbox-123456789012-us-east-2-an",
                    "ObjectKeyPrefix": "inbound/"
                }
            },
            {
                "LambdaAction": {
                    "FunctionArn": "arn:aws:lambda:us-
east-2:123456789012:function:email2webhook",
                    "InvocationType": "Event"
                }
            }
        ]
    }'
```

5. Allow SES to invoke the Lambda function (this is the resource policy from the permissions

table above; the CLI doesn't add it for you the way the console's receipt rule editor does):

```
$ aws lambda add-permission \
    --function-name email2webhook \
    --statement-id AllowSESInvoke \
    --action lambda:InvokeFunction \
    --principal ses.amazonaws.com \
    --source-account 123456789012
{
    "Statement":
"{\"Sid\":\"AllowSESInvoke\",\"Effect\":\"Allow\",\"Principal\":{\"Service\":\"ses.ama
zonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":\"arn:aws:lambda:us-
east-2:123456789012:function:email2webhook\",\"Condition\":{\"StringEquals\":{\"AWS:So
urceAccount\":\"123456789012\"}}}"
}
```

6. Attach `s3:GetObject` to the Lambda's execution role (the third permission from the table — never added automatically). Save this as `s3-read-policy.json`, replacing `<BUCKET_NAME>` and, if you set `SES_S3_KEY_PREFIX`, restricting `Resource` to that prefix:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReadEmailFromS3",
            "Effect": "Allow",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::<BUCKET_NAME>/inbound/*"
        }
    ]
}
```

Then attach it to your function's execution role (no output on success):

```
$ aws iam put-role-policy \
    --role-name email2webhook-role \
    --policy-name AllowReadEmailFromS3 \
    --policy-document file://s3-read-policy.json
```

If this permission is missing, invocations fail with an `AccessDenied` error on `s3:GetObject`, visible in the function's CloudWatch Logs (`/aws/lambda/<function-name>`).

Object retention (lifecycle rule)

Raw emails accumulate in the bucket forever unless something expires them — nothing in this function ever deletes an object (deliberately: it's what makes "re-run this email through the webhook

manually" possible after a failure). Deletion is handled entirely by an **S3 Lifecycle rule** on the bucket, not by this function or its IAM role — it needs no permission on the Lambda's execution role at all.

7. Add a lifecycle rule that expires objects after some number of days. Save this as `lifecycle.json`, adjusting the prefix and retention period:

```
{
    "Rules": [
        {
            "ID": "Delete after 30 days",
            "Filter": { "Prefix": "inbound/" },
            "Status": "Enabled",
            "Expiration": { "Days": 30 }
        }
    ]
}
```

Then apply it (no output on success):

```
$ aws s3api put-bucket-lifecycle-configuration \
    --bucket email-inbox-123456789012-us-east-2-an \
    --lifecycle-configuration file://lifecycle.json
```

Pitfall to avoid if you configure this by hand in the console: use the **Expiration** action (deletes the current, only version of an object after N days), not **"Permanently delete noncurrent versions"** (`NoncurrentVersionExpiration`). The latter only deletes *old* versions of an object after it's been overwritten, which requires bucket versioning to be enabled *and* the same key to be written more than once — neither is typically true here (versioning off, every email gets a unique key), so a rule using it silently deletes nothing, forever, with no error to notice.

`AMAZON_SES_SETUP_NOTIFICATION` is a one-time test object SES writes into the bucket (respecting the configured prefix) when the S3 receipt rule action is first configured, to confirm it can write there — it plays no role in actually receiving email and is safe to delete manually at any time. Left alone, the lifecycle rule above expires it automatically along with everything else, no special handling needed.

Lambda timeout

The function's **Timeout** setting needs enough room for one S3 `GetObject` call plus one HTTP POST to the webhook — the webhook call alone has a 10-second internal timeout in the code (`requests.post(..., timeout=10)` in `send_webhook`). The runtime default of 3 seconds is easy to exceed once S3 latency and the webhook round-trip are added together; a run that happens to finish just under 3 seconds is passing by luck, not by design. Set this to **30 seconds** for a comfortable margin — there's no cost downside to a larger timeout, Lambda only bills for actual execution time:

```
$ aws lambda update-function-configuration \
    --function-name email2webhook \
    --timeout 30
```

Environment variables

Lambda function configuration

Set these on the Lambda function's configuration, for example:

```
$ aws lambda update-function-configuration \
    --function-name email2webhook \
    --environment '{
        "Variables": {
            "SES_S3_BUCKET": "email-inbox-123456789012-us-east-2-an",
            "SES_S3_KEY_PREFIX": "inbound/",
            "WEBHOOK_URL": "https://example.com/webhooks/email2webhook",
            "WEBHOOK_SECRET": "replace-with-a-real-secret",
            "WEBHOOK_FORMAT": "postal",
            "WEBHOOK_SIGNATURE": "hmac"
        }
    }'
```

- `SES_S3_BUCKET` — name of the S3 bucket where the receipt rule stores raw emails.
- `SES_S3_KEY_PREFIX` — optional object key prefix, only needed if the S3 action's "Object key prefix" field was set to a non-empty value; it must match that value exactly, **including the trailing slash** if you want it to appear as a folder in the S3 console (e.g. `inbound/`), since S3 has no real folders — a `/` in the key is just a display convention (default: none).
- `WEBHOOK_URL` — URL the envelope is POSTed to.
- `WEBHOOK_SECRET` — shared secret used to sign the request body when `WEBHOOK_SIGNATURE=hmac` (the default).
- `WEBHOOK_FORMAT` — `postal` (default) or `ses`. See "Output formats" above.
- `WEBHOOK_SIGNATURE` — `hmac` (default) or `none`. See "Signing" above.
- `WEBHOOK_SIGNATURE_PREFIX` — literal string prepended to the hex digest in `X-Webhook-Signature-256` when `WEBHOOK_SIGNATURE=hmac` (default: none, just the hex digest).
- `ATTACHMENT_REQUIRE_EXTENSIONS` — comma-separated list of attachment extensions (with or without the leading dot, case-insensitive, e.g. `xml` or `.xml,.txt`). If set, an email is only processed (and sent to the webhook at all) when it has at least one attachment matching one of these extensions; otherwise it's discarded silently. Applies to both `WEBHOOK_FORMAT` values. Default: empty, no filtering.
- `ATTACHMENT_ALLOW_EXTENSIONS` — same syntax as above. If set, only matching attachments are included in the `postal` format's `attachments` list (non-matching ones are dropped, but the email itself is still sent); has no effect on the `ses` format, which embeds the raw email as-is. Default: empty, all attachments included.

Local testing only

Never set these in the deployed Lambda function; they exist only to run `lambda_function.py` directly on a workstation, without AWS credentials or a live webhook:

- `SES_LOCAL_TEST_EML_FILE` — path to a local `.eml` file; when set, `get_raw_email` reads from this file instead of calling S3.
- `SES_LOCAL_TEST_MESSAGE_ID` — `messageId` value used to build the mock SES event in the `__main__` block (default: `local-test-message-id`).

Deploying

`make all` installs the runtime dependencies listed in `pyproject.toml`'s `[project.dependencies]` into `function/` (`clean` runs first, so no stray files from a previous build or a local test run end up bundled), then zips `function/` into `aws-lambda-email2webhook.zip`. `make clean` removes the zip and everything under `function/` except `lambda_function.py`, restoring it to its checked-in state. `make dev` creates a local `.venv` with the `dev` extras (`ruff`, `mypy`, `boto3`, `requests`, plus type stubs) for linting, type-checking and running the function locally — none of that ever gets bundled into the zip.

`boto3` is deliberately not in `[project.dependencies]` — the Lambda Python runtime already provides it, adding it would only bloat the deployment package. It's only listed under the `dev` extra, to have it available locally for type-checking and for running `lambda_function.py` directly (see "Local testing only" above).

The Makefile pins `python3.14` to build the package, matching the Lambda runtime's version, so there's no risk of a compiled extension (`requests`'s `charset_normalizer` is one) being built against a mismatched Python/OS ABI.

Once you have the zip, upload it — the first time, create the function; afterwards, update its code:

```
$ aws lambda update-function-code \
    --function-name email2webhook \
    --zip-file fileb://aws-lambda-email2webhook.zip
```

Verifying the webhook signature

The receiving end must recompute `hmac_sha256(WEBHOOK_SECRET, raw_request_body)` and compare it (constant-time) against the value in `X-Webhook-Signature-256`. By default that value is just the hex digest, nothing to strip; if `WEBHOOK_SIGNATURE_PREFIX` is set, strip that literal string from the front before comparing.

Last updated on 22/09/2026