Website Difference Checker
Design Document (AWS + Terraform + Python)

Version 1.0 • December 14, 2025

This document specifies a minimal, low-noise website change detector for fewer than 5 URLs checked twice daily. It is intended to be handed to an autonomous implementation agent. The stack is AWS (EventBridge Scheduler, Lambda, DynamoDB, SES, CloudWatch) managed via Terraform, with a Python runtime.

Parameter	Target
Number of URLs	<5
Check frequency	Twice daily (2 runs/day)
Change signal	Content changed (text-based hashing; optional selector scoping)
Notifications	Email via Amazon SES (single digest per run)
State storage	DynamoDB (per-URL hash + timestamps)
IaC	Terraform
Runtime	AWS Lambda Python 3.12 (or latest supported)

Table of Contents

1. Problem Statement and Goals

Build a lightweight service that checks a small set of web pages (fewer than 5) twice daily and notifies the user by email when meaningful page content changes. The service must be inexpensive, low-maintenance, and resilient to minor page noise (e.g., timestamps, cookie banners). It should provide enough context in the email to quickly understand what changed.

Goals

- Twice-daily checks driven by a managed scheduler (no always-on compute).
- Per-URL change detection using normalized text hashing; optional per-URL selector scoping.
- Single email digest per run listing all URLs that changed.
- Persist last-seen state and timestamps in DynamoDB.
- Infrastructure defined in Terraform; app code in Python.

Success Criteria

Over a 30-day period, the service should (a) run on schedule with no manual intervention, (b) avoid frequent false positives, and (c) deliver timely email notifications when the underlying content changes.

2. Non-Goals

- Real-time monitoring or sub-hourly checks.
- Full browser rendering and pixel-level visual diffs (can be a future enhancement).
- User-facing web UI. Configuration is via environment variables and/or a small config file in the Lambda package.
- Complex per-site authentication flows (SSO, cookies, etc.).

3. High-Level Architecture

EventBridge Scheduler triggers a single Lambda function twice daily. The Lambda fetches each URL, normalizes the content, computes a hash, compares it to the previously stored hash in DynamoDB, and sends a digest email via SES if any URL changed. CloudWatch captures logs and metrics; alarms notify on repeated failures.

Components

- EventBridge Scheduler: cron-based triggers (morning and evening).
- AWS Lambda: Python function that performs fetch, normalize, hash, compare, update state, and notify.
- DynamoDB: state table keyed by URL storing last_hash and timestamps.
- Amazon SES: sends digest emails from a verified identity.
- CloudWatch Logs + Metrics + Alarms: operational visibility and alerting.

Data Flow

1 Scheduler invokes Lambda with an event payload containing run_id and timestamp (optional).

2 Lambda loads configuration (list of URLs, ignore rules, email settings).

3 For each URL: fetch HTML, extract text, normalize, compute hash, compare to DynamoDB.

4 If changed: update DynamoDB item, add to digest change list.

5 If any changes: send one SES email containing a per-URL summary and an excerpted diff.

4. Data Model

Use a single DynamoDB table with one item per URL. Primary key is the URL string.

DynamoDB Table: website_diff_state

Attribute	Type	Description
pk (url)	S	Partition key; canonical URL string.
last_hash	S	SHA-256 of normalized content.
last_checked_at	S	ISO-8601 timestamp of last successful check.
last_changed_at	S	ISO-8601 timestamp when hash last changed.
last_notified_at	S	ISO-8601 timestamp of last email notification for this URL (optional cooldown).
etag	S	Last HTTP ETag observed (optional; used for conditional requests).
last_modified	S	Last-Modified header (optional).
error_count	N	Consecutive failure count for this URL (optional).
last_error	S	Last error message (truncated) (optional).

5. Lambda Behavior and Diffing Strategy

Fetch

For each URL, perform an HTTP GET with a stable User-Agent. Use short timeouts (e.g., connect 3s, read 10s) and enable redirects. If stored, send conditional headers (If-None-Match, If-Modified-Since) to reduce bandwidth. Treat 304 as 'unchanged'.

Normalize and Extract

Default strategy is text-based: parse HTML, remove script/style/noscript, extract visible text, collapse whitespace, and optionally remove known noisy patterns (cookie banners, 'last updated' lines). Compute SHA-256 of the normalized text.

Optional Selector Scoping

Support an optional per-URL CSS selector (e.g., 'main' or '#content') to scope extraction when pages have persistent noise. If selector is set and yields content, hash only that region's text; otherwise fall back to full page.

Diff Snippet

For notifications, compute a short human-friendly diff between previous and current normalized text using a line-based algorithm. Include only the top N changed lines (e.g., 20) to keep emails readable. Store full previous text only if you need deep diffs; for this MVP, store only hash in DynamoDB and keep the previous normalized text in S3 (optional enhancement).

Idempotency

A single run is deterministic. Updates are per URL. If the Lambda is retried, the same hash comparison will prevent duplicate 'changed' detection once state is updated. For email, use a per-URL cooldown window via last_notified_at if needed.

6. Notification Content and Anti-Spam Controls

Digest Email Format

Send one email per run if any URL changed. Subject example: 'Website changes detected (2 of 5) - 2025-12-14 AM'. Body includes: run timestamp, list of changed URLs, per-URL excerpted diff, and a link to the URL.

Anti-Spam Controls

- Per-URL cooldown: do not notify more than once within X hours (e.g., 6h) unless explicitly disabled.
- Optional 'confirm on next run': require change to persist across two runs before notifying (off by default for twice-daily checks).
- Truncate diffs and strip long lines to avoid huge emails.

7. Terraform Infrastructure Specification

Terraform provisions: IAM role/policies, Lambda function, EventBridge Scheduler schedules + permissions, DynamoDB table, CloudWatch log group and alarms, and SES identity configuration (note: SES domain verification may require manual DNS steps).

AWS Resources

- aws_dynamodb_table.website_diff_state
- aws_iam_role.lambda_role and aws_iam_policy.lambda_policy (least privilege)
- aws_lambda_function.website_diff_checker (Python runtime, env vars, timeout 30s)
- aws_cloudwatch_log_group for Lambda logs with retention (e.g., 30 days)
- aws_scheduler_schedule.morning and .evening (or aws_cloudwatch_event_rule/target)
- aws_lambda_permission to allow scheduler to invoke Lambda
- aws_ses_email_identity or aws_ses_domain_identity (plus optional DKIM resources)
- aws_cloudwatch_metric_alarm for Lambda errors (optional but recommended)

Terraform Inputs (variables)

- project_name (string)
- region (string)
- schedule_morning_cron (string)
- schedule_evening_cron (string)
- urls (list(string)) or config_s3_uri (string)
- ses_from_address (string)

- ses_to_addresses (list(string))
- ddb_table_name (string, default derived)
- log_retention_days (number, default 30)

Environment Variables (Lambda)

- URLS_JSON: JSON array of URLs (or S3_CONFIG_URI to load config).
- SES_FROM: verified from address.
- SES_TO: comma-separated recipients.
- DDB_TABLE: table name.
- USER_AGENT: custom UA string.
- COOLDOWN_HOURS: optional (default 0).
- IGNORE_REGEX_JSON: optional regex patterns to drop lines.
- SELECTOR_MAP_JSON: optional map of url -> css selector.
- LOG_LEVEL: INFO|DEBUG.

Terraform Packaging Strategy

Use a local build step to create a deployment artifact (zip) containing Lambda code and dependencies. For Python dependencies, either vendor them into the zip (pip install -t build/) or use a Lambda Layer. Given the small footprint, vendoring into the function zip is simplest.

8. Security and IAM

Principles

- Least privilege IAM for Lambda: DynamoDB GetItem/PutItem/UpdateItem on the table; SES SendEmail/SendRawEmail; logs:CreateLogStream/PutLogEvents.
- No secrets required for this MVP. If adding authenticated endpoints later, store secrets in SSM/Secrets Manager and restrict access.
- Avoid placing Lambda in a VPC unless required (keeps egress simple and reduces cold-start overhead).

SES Considerations

If SES is in sandbox, both sender and recipient identities must be verified. For production use, request SES production access. Use domain + DKIM where possible to improve deliverability.

9. Observability and Operations

Logging

Log per URL: status code, whether content was unchanged/changed, normalization strategy used, and elapsed time. Do not log full page content by default. Truncate errors to keep logs readable.

Metrics and Alarms

At minimum, alarm on Lambda invocation errors and on repeated per-URL failures (tracked via error_count in DynamoDB). Optionally publish custom CloudWatch metrics: urls_checked, urls_changed, urls_failed.

10. Testing Plan

- Unit tests for normalization (HTML -> normalized text) using fixed fixtures.
- Unit tests for hashing and diff snippet generation.
- Integration test: run Lambda locally against a known static page (or a mocked HTTP server).
- Deployment smoke test: invoke Lambda manually and confirm DynamoDB updates and SES email delivery.

11. Runbook and Failure Modes

Common Failures

- 403/429 responses: implement backoff, consider reducing frequency or adding headers; mark as failure and continue.
- HTML noise causing false positives: add ignore regex patterns or a selector for that URL.
- SES sandbox restrictions: verify identities or request production access.
- Timeouts: increase Lambda timeout modestly (e.g., 30s) and enforce per-request timeouts.

Operational Steps

- To add a URL: update URLS_JSON (or config file), deploy; optionally seed DynamoDB item on next run.
- To reduce noise: update SELECTOR_MAP_JSON or IGNORE_REGEX_JSON and redeploy.
- To investigate an alert: open email, compare diff snippet, optionally run Lambda in DEBUG to inspect extracted text length and hash.

12. Implementation Checklist for Autonomous Agent

- Create Terraform module with variables listed in Section 7; implement resources and outputs.
- Implement Python Lambda handler: config load, fetch w/ conditional headers, normalize, hash, DynamoDB compare/update, digest email.
- Add requirements.txt and build script to produce zip artifact; wire filename into Terraform aws_lambda_function.
- Configure SES identity and verify sender/recipient as needed; validate SES region.
- Create two schedules (AM/PM) and attach Lambda permission.
- Add CloudWatch log retention and at least one alarm on Lambda Errors metric.
- Provide README with deploy steps and how to update URLs/ignore rules.

Appendix A. IAM Policy Skeleton (Least Privilege)

```
{
"Version": "2012-10-17",
"Statement": [
{
"Sid": "DynamoState",
"Effect": "Allow",
"Action": ["dynamodb:GetItem","dynamodb:PutItem","dynamodb:UpdateItem"],
"Resource": ""
},
{
"Sid": "SendEmail",
"Effect": "Allow",
"Action": ["ses:SendEmail","ses:SendRawEmail"],
"Resource": "*"
},
{
"Sid": "Logs",
"Effect": "Allow",
"Action": ["logs:CreateLogGroup","logs:CreateLogStream","logs:PutLogEvents"],
"Resource": "*"
}
]
}
```

Appendix B. Terraform Sketch (Key Resources)

```
# Sketch only - autonomous agent should expand into full Terraform module

resource "aws_dynamodb_table" "website_diff_state" {
name = var.ddb_table_name
billing_mode = "PAY_PER_REQUEST"
hash_key = "url"

attribute {
name = "url"
type = "S"
}
}

resource "aws_lambda_function" "checker" {
function_name = "${var.project_name}-checker"
role = aws_iam_role.lambda_role.arn
handler = "app.lambda_handler"
runtime = "python3.12"
filename = var.lambda_zip_path
timeout = 30

environment {
variables = {
DDB_TABLE = aws_dynamodb_table.website_diff_state.name
SES_FROM = var.ses_from_address
SES_TO = join(",", var.ses_to_addresses)
URLS_JSON = jsonencode(var.urls)
}
}
}

resource "aws_scheduler_schedule" "morning" {
```

```
name = "${var.project_name}-morning"
group_name = "default"
schedule_expression = var.schedule_morning_cron

flexible_time_window { mode = "OFF" }

target {
arn = aws_lambda_function.checker.arn
role_arn = aws_iam_role.scheduler_invoke_role.arn
}
}
```

Appendix C. Python Pseudocode (Handler Outline)

```
def lambda_handler(event, context):
urls = load_urls()
changes = []
for url in urls:
resp = fetch(url, conditional_headers_from_state(url))
if resp.status_code == 304:
update_checked_at(url)
continue

text = normalize(extract_text(resp.text), url)
new_hash = sha256(text)

prev = ddb_get(url)
if prev is None or prev['last_hash'] != new_hash:
ddb_update(url, new_hash, now)
changes.append(make_change_record(url, prev, text))

else:
ddb_touch(url, now)

if changes:
send_digest_email(changes, now)
```