ChatGPT

Compliance Screenshot Archiver – Design Document

Overview

The Compliance Screenshot Archiver is a micro-SaaS product designed to automatically capture and archive website screenshots (or PDFs) at scheduled intervals for compliance and record-keeping purposes. Many industries (finance, government, etc.) require retaining immutable records of web content changes over time. Our solution will periodically **capture webpages as high-fidelity screenshots/PDFs, store them securely with cryptographic hashes for tamper-proofing, and provide easy retrieval** via a web dashboard and API. The design emphasizes simplicity and cost-effectiveness – leveraging cloud services (AWS) and efficient tools – so a solo developer can build and maintain it. Ultimately, this system will ensure that organizations can **"set and forget" automated web content archiving** and later prove content authenticity when needed.

MVP Scope and Features

The Minimum Viable Product will focus on core features to deliver immediate value:

- **Automated Screenshot Capture (Scheduling):** Users can schedule recurring captures of specific URLs (e.g. daily, weekly). The system will automatically take screenshots or PDFs of the target pages on schedule [1]. This ensures no manual effort is needed to collect records over time.
- **PDF Export of Pages:** Each capture can be saved as a PDF document (or an image) for portability and easy sharing. The system should generate **"defensible" quality PDFs** of the page content [1], preserving the exact appearance of the page at capture time. We leverage headless browser capabilities (Chrome/Chromium) to print the full page to PDF for fidelity [2] [3].
- **Hash Validation (Integrity):** For every captured file, the system computes a cryptographic hash (e.g. SHA-256) and stores it alongside the capture. This hash acts as a **digital signature to authenticate the archived content** [1] – if the file is later altered, the hash verification will fail. This provides tamper-evidence and compliance defensibility.
- **Dashboard (Web UI):** A secure web dashboard allows users to manage their archive. Users can configure capture schedules, view a list of archived screenshots/PDFs, download or preview captures, and verify hashes. The UI will emphasize clarity and simplicity, showing timestamps, target URLs, and hash status for each record.
- **API Access:** All functionality is also exposed via a RESTful API (e.g. for programmatic access or integration into other systems). API endpoints will allow creating/updating capture schedules, triggering an on-demand capture, listing archived files, and retrieving a capture (or its metadata). This enables integration with other compliance systems or automation scripts.

(Out of scope for MVP are advanced features like content change detection, text indexing/search in archives, user roles/permissions beyond basics, or long-term analytics. The MVP will focus on single-user core archiving capabilities, but the design will accommodate future expansion.)

Technology Stack and Design Choices

Backend: We choose **Python** for the backend, using a modern web framework (such as FastAPI or Flask) to build the API server. Python is chosen due to its rich ecosystem and libraries for web automation (critical for our screenshot engine). The backend will be responsible for API logic, scheduling jobs, generating screenshots, computing hashes, and interacting with cloud services. Python's strong support for concurrency (asyncio) and tasks fits well with controlling headless browsers for screenshots.

Screenshot Engine: A custom screenshot capture engine will be implemented using a headless browser automation library. The preferred choice is **Playwright (Python)** or **Puppeteer (via Pyppeteer)** for controlling a headless Chromium browser. Playwright offers robust, high-speed page rendering with support for multiple browsers and asynchronous operation for efficiency [4] [5]. By using a headless Chrome/Chromium, we can ensure that modern web pages (with JS, CSS) render accurately. The engine will navigate to the target URL, wait for the page to load completely (with options to wait for network idle or a fixed delay), then either take a full-page screenshot or directly save the page as PDF. This approach leverages battle-tested browser engines, ensuring reliability in capturing even dynamic content. Both **screenshots and direct PDF generation** are supported by headless Chrome's DevTools protocol (e.g. Chrome's `--print-to-pdf` flag or Puppeteer's `page.pdf()` API) [2] [3]. We prefer generating PDF directly for full-page captures, but the system can also capture PNG images when needed (for thumbnail previews or if image format is preferred). The screenshot engine is optimized to be **cost-efficient** – for example, using **serverless execution** (discussed below) to avoid running a browser 24/7, and possibly reusing browser instances for multiple captures in sequence when feasible, to amortize startup cost.

Frontend: The frontend will be a **React** single-page application, using **ShadCN UI** components for a modern, consistent design. *ShadCN/UI* is a set of open-source, prebuilt React components built with Tailwind CSS and Radix UI primitives [6]. This gives us accessible, themeable components out-of-the-box, speeding up development of the dashboard's forms and tables. The interface will include pages for: managing scheduled URLs (CRUD for schedules), listing archived captures (with filters by date/URL), and a detail view to see a particular screenshot/PDF along with metadata (timestamp, hash value). We will use React with modern hooks and context for state management. Styling will leverage Tailwind utility classes (as ShadCN is Tailwind-based) for rapid UI development and consistency with any design system. The frontend will communicate with the backend via REST API calls (likely using `fetch` or Axios). We aim for a responsive design so the dashboard can be used on various devices if needed (though primary use is desktop). For build tooling, Vite or Create React App can be used to bundle the app, and it can be deployed as a static bundle (served via AWS S3/CloudFront or via the backend).

Cloud Infrastructure (AWS): The solution will be deployed on AWS for reliability and scalability. Key services and architecture decisions include:

- **Compute for Backend & Jobs:** To keep costs low and operations simple, we favor a **serverless architecture**. The backend API can be deployed on **AWS Lambda** using Amazon API Gateway to expose the HTTP endpoints. AWS Lambda's on-demand execution model aligns well with our usage pattern – the service will mostly be idle until a scheduled job runs or a user interacts, at which point a Lambda can execute the needed logic. Heavy tasks like launching a headless browser can be containerized and run in Lambda with sufficient memory. AWS has documented patterns for running headless Chrome (Puppeteer) in Lambda using container image support [7] [8]. This means we can package Chromium and our Python code in a Docker container deployed as a Lambda; when triggered, it launches Chrome in headless mode, captures the screenshot, and then terminates – we **"pay only on demand"** for these short-running tasks [9]. This is extremely cost-effective compared to running a server 24/7 for periodic jobs. Alternatively, we could deploy the backend on a small **EC2 instance or ECS Fargate container** for simplicity (especially if using a framework like FastAPI with Uvicorn). But for the MVP, we will attempt a Lambda-based approach for the actual screenshot capture process to maximize cost savings and scalability (each capture can scale as a separate lambda, enabling parallel captures when many are scheduled) [10]. The API routes and scheduling logic could also be in Lambda or a lightweight container; using API Gateway + Lambda for all API endpoints will eliminate server management.

- **Storage:** Captured screenshots/PDF files will be stored in **Amazon S3**, which is ideal for durable, scalable object storage. Each capture file will be saved to an S3 bucket, organized by user and timestamp (e.g. `s3://compliance-archiver/{user_id}/{site_name}/{timestamp}.pdf`). S3 provides 99.999999999% durability and lifecycle management (useful if we later want to expire or archive older captures). All files will be stored with server-side encryption (SSE) for security. Storing in S3 also makes retrieval easy – the dashboard can fetch a file via a presigned URL or through the backend as needed. S3 is already used in similar solutions to hold screenshots taken by serverless functions [11]. Additionally, S3 can store metadata or thumbnails if needed, but metadata will primarily reside in a database (see below).
- **Database:** We need a database to store metadata about each capture and the user's configurations. For MVP, the data schema is relatively simple (e.g. *Users*, *Websites*, *Schedules*, *Captures*). We have two good options on AWS: a **relational DB (RDS)** or a **NoSQL DB (DynamoDB)**. Given the simplicity of relationships (each user has many scheduled websites, each schedule produces many capture records), a relational model is straightforward. We may use **PostgreSQL on Amazon RDS** for familiarity and the ability to use SQL for queries (e.g. listing captures between dates). However, using Amazon DynamoDB (NoSQL) could simplify scaling and ops (no maintenance), at the cost of more complex query logic for certain filters. DynamoDB is very attractive for serverless deployments since it's fully managed and can store items with flexible schema (each capture item could include the URL, timestamp, S3 key, hash, etc.). In fact, the AWS reference architecture for screenshot pipelines uses DynamoDB to store metadata linking S3 object keys with timestamps and user IDs [12] [13]. For MVP, we lean towards **DynamoDB** for its serverless nature and simplicity: we'll have a table for Schedules and a table for Captures. Each Capture record will include a primary key (capture ID or composite of schedule+timestamp), the capture timestamp, the target URL, a reference to the S3 file (object key), and the stored hash. Schedules table will store the user's schedule definitions (with fields for URL, cron expression or interval, last run, etc.).
- **Scheduling Mechanism:** To implement the scheduling of screenshot jobs, we have a couple of approaches:
- **Cron-based Scheduler (App level):** Run a scheduler process that checks due tasks and triggers captures. For example, if the backend runs on a container or EC2, we could use a library like *Celery Beat* or APScheduler to schedule tasks according to the user-defined cron expressions. This scheduler would enqueue a job (possibly via Celery workers or an internal task queue) to run the screenshot capture. However, in a fully serverless setup (where we might not have a constantly running server process), we can't have an in-memory scheduler always on.
- **AWS EventBridge Scheduler:** AWS offers managed cron scheduling via EventBridge (formerly CloudWatch Events). We can programmatically create an EventBridge rule for each user schedule (with a cron expression) that triggers a Lambda function at the specified times. The Lambda would then invoke the screenshot capture logic for that particular schedule. This approach offloads scheduling to AWS – which is reliable and scalable – but could become unwieldy if there are a very

large number of schedules (many distinct rules). Alternatively, a single EventBridge rule can trigger a master Lambda every minute, which then checks the database for any schedules due in that minute and triggers the captures. For simplicity, we might start with **one periodic scheduler Lambda**: e.g., every 5 minutes it runs and triggers due capture tasks (invoking the capture Lambda for each needed URL). This is a cloud-friendly equivalent of a cron job. Each capture Lambda will run independently, allowing parallel captures if multiple schedules align at the same time [10]. This design ensures that adding more schedules scales linearly and we pay very little when there are no jobs (just a quick check query on the DB periodically).
- **API & Networking:** If using Lambda + API Gateway, the REST API endpoints (for dashboard and external use) will be defined in API Gateway, each mapping to a Lambda function (or a unified Lambda that uses a framework like FastAPI inside). Endpoints include: user authentication, CRUD for schedules, trigger immediate capture, list captures, get capture file URL, etc. We will secure the API (using AWS Cognito for user sign-up/login would be ideal, or a simple JWT system for MVP). API Gateway can integrate with Cognito user pools to handle auth easily [14]. For the front-end, if it's hosted as a static site (e.g. S3 + CloudFront), it will communicate with the API Gateway endpoints via HTTPS calls. We will enable CORS on the API for the dashboard's domain.
- **Notifications (optional for MVP):** While not required in MVP, AWS SNS or email could be used to notify users after each capture (or on failures). For now, the dashboard will surface status, but we note the potential of integrating Amazon SNS or Amazon SES if email alerts for capture results are needed in future.

System Architecture: At a high level, the system is split into three layers: the frontend app, the backend services (API + screenshot engine), and the storage layer (database and file storage). Below is an overview diagram of how these components interact:

High-level architecture of the Compliance Screenshot Archiver. The React/ShadCN frontend (client) communicates with the backend API (Python on AWS Lambda or container). When a scheduled time arrives, a Scheduler component (EventBridge or a cron Lambda) triggers the Screenshot Engine (headless browser running in AWS Lambda) to capture the target webpage. The captured page is saved to AWS S3 as a PDF/image, and metadata (URL, timestamp, hash, etc.) is stored in the Database (DynamoDB). The dashboard and API clients can retrieve the list of archives (from DB) and download files from S3 (often via a presigned URL or CloudFront CDN). All secrets and configuration (e.g. AWS keys, DB connection) are managed via AWS Parameter Store or Secrets Manager.

(Note: The diagram is conceptually based on AWS's serverless reference architectures for image processing [12] [13] and web scraping with Puppeteer [10], adapted to include our scheduling and database components.)

Frontend Design (React & ShadCN UI)

The frontend will be implemented as a **React** application using **TypeScript** for robustness. Using the ShadCN UI library, we will import ready-made components for common UI elements such as forms, modals, tables, dropdowns, etc. ShadCN provides accessible and themeable components built on Tailwind and Radix primitives [6], meaning we can easily customize the styling via Tailwind CSS while benefiting from Radix's accessible behaviors (keyboard navigation, focus management, etc.).

Key UI screens and elements:
- **Login/Signup**: A simple authentication page (if we don't offload auth to AWS Cognito's hosted UI). Users will log in to access their dashboard. Authentication tokens (JWT) will be stored (probably in memory or

secure cookies, not localStorage for security). If using Cognito, we might integrate their JS SDK or use a hosted UI redirect flow for simplicity.
- **Dashboard Home**: Overview of the service status – possibly a summary of upcoming scheduled captures and recent captures. Could show a calendar view of what runs have happened or are scheduled.
- **Schedule Management**: A page where users can add a new URL to archive. A form will allow inputting the target URL, selecting a schedule frequency (with options like daily/weekly or a cron expression input for advanced). We will validate the URL and schedule format on the frontend for usability. The schedules list will be displayed, each with actions to edit or delete.
- **Archive View**: A page listing all captured records (perhaps in a table with columns: Timestamp, URL (or a friendly label), status, and actions). The user can filter by URL or date range. Each record row will have an action to "View" or "Download" the capture.
- **Capture Detail/Preview**: When viewing a specific capture, we can show a PDF preview (if feasible in-browser via PDF.js or an `<iframe>` if the PDF is accessible), or an image preview if it's an image. We will also display the metadata: capture time, original URL, the SHA-256 hash, and perhaps a copy button to copy the hash. We may also show a verification status (by re-computing the hash of the file on-demand and comparing with stored hash, to confirm integrity). For PDF, the hash is computed on the PDF binary; if image, on the image file.
- **Responsive Design**: We will ensure the layout works on common screen sizes. Tailwind utilities and ShadCN components will help in creating a responsive grid or stack layout. However, primary usage is likely on desktop by compliance officers.
- **Error Handling & UX**: The UI will handle cases like a capture failing (we can mark a capture entry as "Failed" with an error message if, say, the site was unreachable). It will also guide the user to provide correct input for schedules (maybe offering a simple schedule builder UI).

For state management, simple React `useState` and `useEffect` hooks may suffice, perhaps lifting state up to context providers for things like auth status. If the app grows in complexity, we might introduce a state management library or React Query for data fetching caching. Initially, straightforward data fetching on component mount (with loading spinners) is fine. All data will come from the backend API, which returns JSON. For file downloads, the frontend might get a presigned URL from the API or an API endpoint that streams the file; then the frontend can initiate the browser download or open the PDF.

We will also include basic input validations and user feedback (using ShadCN's form components to display validation errors nicely). Since this is a compliance tool, **accuracy is more important than flashy design** – we will prefer a clean, minimalistic interface that clearly presents data and statuses.

Backend Design (Python API & Services)

The backend consists of a RESTful API and the background processing logic for captures. Key components of the backend design include:

1. Web API (RESTful): We will implement a set of endpoints to cover all operations. For example:
- `POST /api/schedules` – create a new scheduled capture (with JSON body containing URL and schedule cron or interval).
- `GET /api/schedules` – list all schedules for the authenticated user.
- `PUT /api/schedules/{id}` – update a schedule (e.g. change frequency or URL).
- `DELETE /api/schedules/{id}` – remove a schedule.
- `POST /api/captures/trigger` – trigger an on-demand capture for a given URL (or an existing

schedule). This returns as soon as the job is queued or started (for synchronous capture, could also wait for result, but async is preferable).
- `GET /api/captures` – list capture metadata (with query params for filtering by URL or date).
- `GET /api/captures/{id}` – get metadata of a specific capture (including the hash and a download link or ID). Possibly also used to fetch the actual file (though we might redirect to S3).
- `GET /api/captures/{id}/download` – (if not using presigned URLs) stream the PDF/image file to the client.
- `GET /api/health` – simple health check.

These endpoints will be protected – requiring an Authorization token (JWT or Cognito identity) to ensure each user only accesses their own data. The API will be stateless. If using API Gateway + Lambda, each request hits a lambda which initializes the app (if using a framework like FastAPI, we might use a lightweight approach to not cold-start too slowly). We can also consider running a single container on ECS with a small Flask/FastAPI app if persistent connections are needed (but likely not required).

2. Screenshot Capture Engine: This is the core module responsible for launching a headless browser and capturing the screenshot/PDF. It will be implemented as a Python function/class that can be invoked with parameters (URL, perhaps options like full-page vs viewport, output format). Key steps in this process:

- Launch a headless browser instance. If using **Playwright**, we will launch Chromium with `headless=True` [15]. If using **Pyppeteer**, we call `launch()` similarly [16]. This can be done on-demand for each capture. In the future, we could maintain a warm browser to reuse, but in a serverless context, that's not persistent, so per-invocation is acceptable.
- Navigate to the target URL and wait for the page to load fully. We will use appropriate waiting strategies (e.g. `await page.goto(url, wait_until="networkidle")` in Playwright) to ensure dynamic content is loaded. Optionally, allow a configurable delay or waiting for a specific DOM element if needed.
- Capture the page:
  - For **PDF mode**, call the browser's PDF generation function (Playwright's `page.pdf()` or Puppeteer's `page.pdf()` which can take options for page format). This produces a PDF file of the whole page as if "printed" [3].
  - For **Screenshot (image) mode**, call `page.screenshot()` to get a PNG image of the full page (we can set `fullPage: true` in Puppeteer or use a loop to scroll and stitch if needed, but headless Chrome supports full page capture natively). Ensure a suitable viewport size or device emulation if required for consistency.
- Save the output file: The Lambda (or server) will then upload the resulting file to S3. If the file is small, the API could even base64-encode and send it back, but storing in S3 is more scalable. The S3 SDK (boto3 in Python) will be used to `put_object` the file into our bucket. We'll include metadata in the object like content-type and perhaps the hash (though we will also store hash in DB). The S3 key can be the unique ID of the capture or a structured path (as mentioned earlier).
- Compute the hash: We will compute a SHA-256 hash of the file bytes using Python's `hashlib`. This is done immediately after capture (before or after upload). This hash string will be stored in the database record for this capture. (If needed, we could also store it as S3 object metadata or even in the PDF file as a custom attribute, but a DB entry suffices).
- Return status: If this capture was triggered by an API call (on-demand), the API can return success once the file is stored (perhaps returning the capture ID or URL). If it's a scheduled capture triggered

in background, no immediate API response is needed; the result will be visible in the dashboard when the user checks.

The screenshot engine must handle errors gracefully – e.g., if the page fails to load or times out, or if the headless browser crashes. We will implement retries (maybe try twice if transient error) and timeouts (don't wait more than X seconds per page). Errors will be logged (CloudWatch logs for Lambda). If a capture ultimately fails, we mark that in the DB (with an error message) so the dashboard can show the capture as "Failed" and possibly allow re-try manually.

We aim for the engine to be **efficient and low-cost**. Running headless Chrome is CPU/RAM intensive, but in AWS Lambda we can allocate more memory which also boosts CPU. We might allocate ~1024MB or 2048MB RAM for the Lambda to ensure Chrome runs quickly; the Lambda's duration will likely be only a few seconds per capture. This means each capture costs a few hundred milliseconds of Lambda time, which at scale is cheap (especially with AWS Free Tier [17]). Additionally, using Lambda means we scale **concurrently** – if multiple captures are scheduled at the same time, multiple instances run in parallel. This parallelism ensures one slow website doesn't block others, and it meets the compliance need of capturing at the scheduled times. (In the future, if some users have dozens of URLs, we could implement a fan-out: one Lambda to fetch schedule info and trigger multiple capture Lambdas in parallel – similar to AWS's example using a fan-out Lambda [10] – but for MVP a simpler direct trigger per schedule is fine.)

3. Scheduling Service: As described in the infrastructure section, the scheduling can be handled by AWS EventBridge or an internal scheduler: we will likely configure an EventBridge rule or CloudWatch event that triggers a **Scheduler Lambda** every N minutes. This Scheduler Lambda will: query the DynamoDB "Schedules" table for any schedules that should run in the current timeslice (for example, it can compute the next run times or store next run timestamps in the table). For each due schedule, it will invoke the Screenshot Engine (either by directly calling the capture Lambda with the target URL and user info as payload, or by publishing a message to an SQS queue that a separate consumer listens to). Using SQS + Lambda triggers is another method: the scheduler could drop a message "capture X now", and a Lambda reading the SQS will perform it. However, invoking the capture Lambda directly is simpler. The scheduler will then update the "last run" timestamp in the schedule record (and compute the next run time if needed, though with cron expressions we can compute on the fly each time).

If we opt for creating individual EventBridge cron triggers per schedule, the design is simpler in code (EventBridge handles timing), but it might be harder to manage a large number of rules and to allow users to update them. Programmatically, we can use the AWS SDK to add a rule with a cron expression and target (the capture Lambda with specific parameters) whenever a new schedule is created. For MVP with a small number of schedules, this is feasible. We must also delete the rule if schedule is removed. This approach tightly couples to AWS and might have limits on number of rules; thus, we lean towards the **central scheduler** lambda approach as it's more generic and easier to keep multi-tenant (one lambda that checks all schedules).

4. Data Model & Storage: We will design our data models as follows (assuming DynamoDB for description, but analogous tables in SQL if using RDS):

- **User**: (If implementing user accounts ourselves) Contains user info, login credentials (if not offloaded to Cognito), etc. Each user gets a unique ID. This could be in a Cognito user pool or a simple table. For MVP, we might rely on Cognito for user management, so no custom user table needed.

- **Schedule**: Fields – ScheduleID (primary key), UserID (to partition by user), URL, CronExpression or Interval, Description (optional label), LastRun (timestamp of last capture), NextRun (if we pre-compute it), Active flag. In DynamoDB, we can have partition key = UserID, sort key = ScheduleID, so we can query all schedules by user easily. In SQL, we'd have a schedules table with a foreign key to user.
- **Capture**: Fields – CaptureID (primary key or a composite of ScheduleID+timestamp), ScheduleID (which schedule it came from, or null if it was ad-hoc capture), UserID, URL (store for redundancy in case schedule is deleted), Timestamp (capture time), S3Key (path to file in S3), FileType (PDF/PNG), Hash (SHA-256 string), Status (Success/Failed), maybe an ErrorMsg if failed. DynamoDB can use partition key = ScheduleID (or UserID) and sort key = Timestamp (for easy range queries by date). We will likely also have a GSI on UserID to get all captures across schedules if needed. If using SQL, a captures table with foreign key to schedule (nullable for ad-hoc) and an index on timestamp.
- We do not store the actual image/PDF in the DB, just its location and hash. This keeps the DB records small. All large data (files) are in S3.

Given the importance of integrity, we'll also ensure the **hash is stored safely**. We might consider also storing a copy of the hash in a secure audit log or even using AWS QLDB or blockchain for absolute tamper-proof logs in future. For MVP, DynamoDB is tamper-evident enough (with restricted access) combined with the hash itself proving file integrity.

5. Hash Verification Process: Whenever a user wants to verify a file (e.g. to demonstrate to an auditor), they can download the file and compute its SHA-256 hash and compare to the one in our system. We could provide a small verification tool or simply let them use any standard tool to compute it. In the dashboard, we might add a "Verify" button on a capture record that triggers the backend to recompute the hash from S3 and compare to the stored hash (to double-check it's still matching). Since S3 files shouldn't change (we never overwrite captures), this is mostly to reassure that nothing got corrupted. This on-demand verification can be done by reading the object from S3 (or using S3's ETag if we stored the hash as ETag by uploading with Content-MD5 or something, but SHA-256 is stronger).

6. Security and Compliance: We will enforce security best practices: all communication is over HTTPS. If multi-tenant, each API call will validate the authenticated user's ID against the resource (to ensure one user cannot access another's data). In AWS, each Lambda will run with least-privilege IAM roles (e.g., a role that only allows it to read/write the specific S3 bucket and the specific DynamoDB tables). S3 bucket will have access control so only the backend (and authorized users via signed URLs) can retrieve files. We will enable server-side encryption on S3 (AES-256 by default, or AWS KMS if needed for extra compliance). DynamoDB can also have encryption at rest (enabled by default). If using Cognito for auth, that simplifies storing user credentials (Cognito will store and handle password policies, MFA if needed, etc.). If not, we will securely hash passwords (if any) in our user table using something like bcrypt.

For compliance, **audit logs** are important. We will log key events (schedule created/modified, captures executed, any errors) possibly to CloudWatch. This can help trace issues and also serve as an audit trail of activities in the system (e.g., showing that a capture was indeed taken at time X or if it failed due to site outage). These logs can be retained as needed.

Workflow Example

To illustrate the end-to-end flow, consider a user who wants to archive `https://example.com` daily:

1. **User Setup:** The user logs into the dashboard and creates a new schedule: enters URL `https://example.com`, selects "daily at 09:00 AM" from a schedule dropdown. The frontend calls `POST /api/schedules` with this data. The backend (API Lambda) creates a new Schedule record in the DB with cron expression `0 9 * * *` and associates it to the user. If using EventBridge per schedule, here we also call AWS SDK to create the cron rule. Otherwise, the schedule just sits in DB.
2. **Scheduler Trigger:** At the next 09:00 AM, the Scheduler lambda (triggered by EventBridge every minute or by the specific rule) finds that this schedule is due. It invokes the Screenshot Engine – likely by calling a Lambda function (e.g. `CaptureLambda`) with payload `{scheduleId: ..., url: "https://example.com", userId: ...}`.
3. **Capture Execution:** The Capture Lambda spins up (cold start if first time, then reuses if warm). It initializes the headless browser (Playwright). It navigates to `https://example.com` and waits until network is idle or a timeout of e.g. 10 seconds. Then it generates a PDF of the page using the browser's API [2]. The PDF bytes are obtained in memory. The lambda computes the SHA-256 hash of those bytes. It then uses boto3 to upload the PDF to S3 bucket `compliance-archiver` at key `user_{UserID}/schedule_{ScheduleID}/2025-08-20T09-00-00.pdf`. The object metadata includes perhaps the ISO timestamp and maybe the hash (for reference). Upon successful upload, it writes a new item to DynamoDB "Captures" table with details: ScheduleID, Timestamp, S3Key, Hash, Status=Success. If any step fails (exception), it instead records Status=Failed and an error message. The lambda returns or exits.

4. **Dashboard Update:** The user may receive a notification or simply check the dashboard later. The dashboard's Archive page calls `GET /api/captures?scheduleId=...` to get all captures for that schedule. The new capture of today appears in the list, with its timestamp and a "success" status. The user can click "View PDF" which triggers either downloading from S3 via a presigned URL or streaming through our API. The user opens the PDF and sees the snapshot of `example.com` as of that date. The dashboard also displays the hash; if the user clicks "verify", the frontend calls `GET /api/captures/{id}/verify`, and the backend lambda will fetch the file from S3, recompute hash, and respond with a match/doesn't match result (should match). The user can also download the PDF and share with auditors, along with the hash value for them to independently verify.

5. **API Access:** If the user had an external system (say a compliance management system) that wanted to fetch the latest capture programmatically, it could call our API with the user's token. For example, `GET /api/captures?url=example.com&latest=true` could be an endpoint to get the latest capture record and a link. This would return JSON with the needed info, which the external system could use to pull the file (if permitted).

6. **Hash and Integrity:** Later, if anyone questions the authenticity of the archived page, the stored hash can be used to prove the file wasn't altered. We might even consider timestamping the hash (e.g., storing it in a blockchain or notarization service) as an enhancement, but MVP will rely on the hash and secure storage alone. The PageFreezer guide to compliance archiving underscores the importance of both hash-based authentication and PDF exports for legal defensibility [1], which is exactly what our system provides.

Future Considerations and Optimizations

While the MVP focuses on a simple, single-user scenario, the design is meant to be extensible:

- **Scalability:** As the number of users or scheduled pages grows, the serverless architecture can scale almost automatically (Lambda will spawn more instances as needed, DynamoDB can handle high throughput with on-demand capacity, and S3 is essentially infinite storage). We will monitor costs, as many frequent captures or very large PDF captures could incur Lambda and storage costs – these can be mitigated by using caching or adjusting schedules to avoid unnecessary frequency. If needed, a pool of headless browser instances on an EC2 auto-scaling group could be introduced for very high volumes (to avoid Lambda cold starts), but this adds complexity and cost (only worth if usage patterns justify it).
- **Optimizing the Screenshot Engine:** We can consider using a lighter-weight renderer if Chrome is too heavy. For example, there are headless browser services or smaller engines like **Splash** (headless browser with an HTTP API) [18] [19]. Splash is resource-light but would require running a persistent service (maybe as a Docker container in ECS) and doesn't support as much JS as Chrome. Another idea is using **wkhtmltopdf** for PDF generation of static pages (it uses Qt WebKit), but it might not handle modern JS as well as Chrome. For now, Chrome/Playwright is the reliable choice. We will also look into reusing the browser between captures in the same Lambda invocation: AWS Lambda allows some reuse if the function stays warm, so our code could be written to keep the browser open and only launch it if not already open (this requires the Lambda to not be forced to quit after one request – with AWS API Gateway one-invocation-per-request, we might not get reuse unless using the capture Lambda for multiple captures via SQS). This optimization might not be needed until high scale.
- **Multi-tenancy & Organization:** With growth, we'd implement organization accounts, multiple users, roles (admin, auditor roles who can only view but not schedule, etc.). Also features like **Legal Hold** (ensuring data isn't auto-deleted during a retention period) are mentioned in compliance guidelines [1] – we can add settings for how long to keep archives. S3 lifecycle policies can auto-archive or delete data after X years if needed.
- **Search and Indexing:** In future, making the archived pages searchable (via OCR or text extraction) could add value. We could store text of pages (perhaps extracting innerText with the browser) in the DB to allow keyword search in the dashboard. However, this is beyond MVP.
- **Visual Change Detection:** Some services compare screenshots to detect changes. We could incorporate a diffing mechanism (pixel or DOM diff) to highlight what changed since last capture, and notify if needed. Again, not core compliance (which usually requires full capture regardless of change), but could be a useful add-on.
- **Alternate Outputs:** Support WARC (Web ARChive format) or HTML snapshot storage for more complete archiving (some argue static screenshots/PDFs are limited [20]). Perhaps far future: store the raw HTML/CSS/JS and use a viewer to reconstruct the page, but that's complex and out of scope for now. Our focus remains on visual snapshots which are often acceptable for evidence, as long as we capture the key content.

In conclusion, this design provides a **simple yet comprehensive architecture** to implement the Compliance Screenshot Archiver. By using a Python-driven headless browser approach and AWS serverless services, we ensure the solution is **reliable (using proven browser engines)**, **efficient (scales on demand, minimal idle cost)**, and **secure (hash validations and cloud security best practices)**. The technology choices (React/ShadCN for UI, Python/Playwright for backend, AWS for infra) align well with rapid development by a solo builder, aided by AI coding tools. This design document will guide the implementation, ensuring that we deliver a functional MVP that meets the core compliance archiving needs and can evolve with future requirements.

Sources: The design draws on established practices in web archiving and serverless automation, including AWS's example of using Lambda+Puppeteer for screenshots [11], compliance archiving guidelines from industry (emphasizing hash-based authenticity and PDF exports) [1], and technical evaluations of headless browser tools (highlighting the performance benefits of modern tools like Playwright) [4]. These informed our choices to build a solution that is both robust and developer-friendly.

---

[1] Complete The Compliance Guide to Archiving Online Data
https://www.pagefreezer.com/the-complete-compliance-guide-to-archiving-of-online-data/

[2] [3] Getting Started with Headless Chrome | Blog | Chrome for Developers
https://developer.chrome.com/blog/headless-chrome

[4] [5] [15] [16] [18] [19] The Best Python Headless Browsers For Web Scraping in 2024 | ScrapeOps
https://scrapeops.io/python-web-scraping-playbook/python-best-headless-browsers/

[6] ShadCN UI vs Radix UI vs Tailwind UI, Which Should You Choose in 2025? | by Piyush zala | Jul, 2025 | JavaScript in Plain English
https://javascript.plainenglish.io/shadcn-ui-vs-radix-ui-vs-tailwind-ui-which-should-you-choose-in-2025-b8b4cadeaa25?gi=74590fe95045

[7] [8] [10] [11] Field Notes: Scaling Browser Automation with Puppeteer on AWS Lambda with Container Image Support | AWS Architecture Blog
https://aws.amazon.com/blogs/architecture/field-notes-scaling-browser-automation-with-puppeteer-on-aws-lambda-with-container-image-support/

[9] [17] Show HN: Chromeless – Headless Chrome Automation on AWS Lambda | Hacker News
https://news.ycombinator.com/item?id=14859084

[12] [13] [14] Serverless In-Game Screenshot Processor Pipeline for Game Studios - Serverless In-Game Screenshot Processor Pipeline for Game Studios
https://docs.aws.amazon.com/architecture-diagrams/latest/serverless-in-game-screenshot-processor-pipeline-game-studios/serverless-in-game-screenshot-processor-pipeline-game-studios.html

[20] Compliance Relics: The Case Against PDFs and Screenshots
https://www.mirrorweb.com/blog/compliance-relics-the-case-against-pdfs-and-screenshots