Project JSync: JIRA Exporter and Synchronization System

by Dinis Cruz and Claude 3.5, 2025/02/05

Project Scope

This project JSync will implement a **serverless JIRA change capture and sync system** for internal use.

The goal is to **capture all changes in JIRA issues** (creations, updates, transitions, etc.) in real-time and store them in two places:

- **Amazon S3** – as JSON snapshots for backup and easy retrieval.
- **GitHub** – as version-controlled records (so we have full history and audit via Git commits).

Key requirements include processing each JIRA update **within seconds** of the event (near real-time syncing). The system will be **API-driven**: we will build a RESTful API (using Python's FastAPI framework) to allow querying the stored JIRA data and managing the sync process.

This API will be documented with **OpenAPI/Swagger**, making it easy for developers to understand and interact with (interactive docs for internal users). Importantly, the entire solution should run with **no persistent servers** – leveraging AWS managed services (Lambda, S3, etc.) for a fully serverless, auto-scaling architecture. This ensures minimal ops overhead and that we only pay for usage.

In summary, the project's scope covers: capturing JIRA issue changes in real-time, transforming and storing those records in S3 and a GitHub repository, exposing an API for access, and doing so with high performance and thorough documentation.

All code will be in Python, and extensive automated tests (targeting **100% coverage**) will be written to ensure reliability.

Architecture & Technology Stack

The system will use a **serverless event-driven architecture** on AWS. The key AWS components and technologies include:

- **AWS Lambda**: Lambda functions form the core compute layer. One Lambda will be triggered by JIRA webhooks (via an HTTP endpoint) to process incoming data. This function will parse JIRA's webhook payload (XML from JIRA, which will be converted to JSON) and then handle persistence (writing to S3, and updating GitHub). We will use **Lambda Container Image** support for the portion that interacts with GitHub, so that we can include a Git client inside the Lambda environment. (Normally, one cannot install arbitrary binaries in Lambda's standard environment (How do I install Git using AWS Lambda? - Stack Overflow), but containerizing the function allows us to have Git installed and any other dependencies needed.) All Lambda functions will be written in Python (using FastAPI for the API endpoint logic).
- **Amazon S3**: S3 will serve as the storage for JSON data representing JIRA issues and their changes. Each JIRA issue change event will result in a JSON file being saved to an S3 bucket (for example, in a key structure like `issues/<PROJECT>/<ISSUE_KEY>/<TIMESTAMP>.json`). We may also choose to keep a single JSON per issue (the latest state) and enable versioning on the bucket so that all historical versions are retained automatically. Storing data in S3 provides durable, cost-effective storage that can be easily accessed by other tools or via the API. S3 will be configured with server-side encryption and access controls to ensure the data is secure.
- **GitHub Repository**: The system will integrate with GitHub to store a second copy of the data. The Lambda function (running in a container with Git) will clone or pull the GitHub repo (which contains JSON files for each issue), commit the new changes (e.g. add or update the JSON for the issue that changed), and push back to GitHub. This provides a version-controlled history of all changes. We will likely organize the repository in a similar fashion to S3 (one file per issue, or per event) so that diffs show the evolution of each issue. The GitHub integration will use a personal access token (or GitHub App credentials) stored in AWS Secrets Manager (see below). Each commit can trigger a GitHub Action as part of CI/CD (for example, to run further checks or to open a pull request if we want manual review for some changes, though for this use-case an auto-commit to a version-control branch is fine).
- **Amazon CloudFront**: CloudFront will be used as a CDN in front of our API and possibly the S3 bucket, to accelerate access and provide a cached layer. We will configure CloudFront with a custom domain (via Route 53) to serve the FastAPI endpoints and the static JSON content. CloudFront can cache GET responses from the API (improving performance for frequently accessed data) and also directly serve the static files from S3 (if we choose to expose them, for example for browsing snapshots). This will also help with global performance if our internal teams are geographically distributed, ensuring low latency access. CloudFront will be configured with appropriate cache behaviors (e.g. dynamic API responses might have

short TTL or no caching, whereas static documentation or rarely-changing content can be cached longer). CloudFront will also be used to connect requests to Lambda functions via AWS Function URL capabilities.

- **Amazon Route 53**: Route 53 will manage the DNS for the service's endpoints. We will create a domain or subdomain (for example, `jira-sync.internal.company.com`) and use Route 53 to route traffic to the CloudFront distribution. This gives a friendly, stable URL for internal users and for configuring the JIRA webhooks. It also allows us to manage SSL certificates (through AWS Certificate Manager) for that domain, so that the API is served securely over HTTPS. Route 53 costs are minimal ($0.50/month per hosted zone and small charges per DNS query) and it integrates well with CloudFront.
- **AWS IAM**: Identity and Access Management will be used to enforce least-privilege access for all components. We will create specific IAM roles for the Lambda functions: for example, the webhook-processing Lambda will have permission to put objects to the S3 bucket, pull from ECR (for container image), write logs to CloudWatch, and read the GitHub token from Secrets Manager. Another IAM role may be used for the FastAPI Lambda if it has different permissions (like only read access to S3). No broad wildcard permissions will be used – everything will be scoped to just what's needed (e.g. specific bucket ARN, specific secret ARN). IAM will also manage any user access if needed (though likely the API will not require AWS IAM auth since it's internal, but we might use IAM roles for services that call the API).
- **Amazon CloudWatch**: CloudWatch will capture logs and metrics from Lambda and CloudFront. Every invocation of the Lambda (for webhook or API calls) will generate logs (including any errors or debugging info) to CloudWatch Logs. We will set log retention policies to a reasonable period (for example, 30 days) to manage costs. CloudWatch will also be used to create custom metrics/alarms: for instance, we can monitor the Lambda invocation failures or duration, and set up an alarm if errors suddenly spike (to alert the team). CloudWatch dashboards can be created to visualize the number of JIRA events processed, processing time, etc. All of this helps in monitoring the health and performance of the system.
- **AWS Secrets Manager**: Secrets Manager will safely store sensitive configuration, notably the **GitHub credentials** (like a Personal Access Token or an SSH key for the repo) and possibly any JIRA credentials (if in the future we need to call the JIRA API, e.g. for a full sync). These secrets will be encrypted and only accessible by the Lambda functions (via IAM policy). The Lambda at runtime will retrieve the secret (Secrets Manager charges $0.05 per 10,000 API calls for secret retrieval, which is negligible (How to rotate a WordPress MySQL database secret using AWS Secrets Manager in Amazon EKS | AWS Security Blog)). The cost of storing secrets is $0.40 per secret per month (How to rotate a WordPress MySQL database secret using AWS Secrets Manager in Amazon EKS | AWS Security Blog) – for example, two secrets would be ~$0.80/month. Using Secrets Manager eliminates hard-coding sensitive info and allows easy rotation of credentials without code changes.

- **AWS Elastic Container Registry (ECR)**: (Implied in the use of Lambda containers) We will use ECR to store the Docker container images for our Lambda functions. Each time we update the Lambda code (especially for the one including Git and possibly the FastAPI app), a Docker image will be built and pushed to a private ECR repository. AWS Lambda will pull the image from ECR when provisioning instances of the function. ECR is a fully managed Docker registry and works seamlessly with Lambda's container support. We will use appropriate base images (Amazon Linux 2 base for Lambda with Python) and install Git and our Python code into that image. IAM roles and permissions for Lambda to access the ECR repo will be configured. (ECR cost is low – $0.10 per GB-month of storage and data transfer out, which in our case is trivial since the image might be a few hundred MBs and seldom pulled).

All these services together form a cohesive, serverless architecture. The design ensures **scalability** (Lambdas scale automatically to handle bursts of events), **durability** (S3 and GitHub provide persistent storage of data), and **security** (using IAM, and Secrets Manager for creds). There are no always-on servers to maintain – everything runs on-demand. The architecture is summarized as follows:

1. **Webhook Ingestion** – JIRA triggers an HTTP POST to our API whenever an issue changes.
2. **Lambda Processing** – The webhook invocation triggers the Lambda function which runs our Python code (FastAPI logic for the endpoint) to parse and handle the data.
3. **Data Storage** – The Lambda writes the JSON to S3 and commits it to GitHub (through the Git-enabled container).
4. **API Access** – Another Lambda (running FastAPI) provides GET/POST endpoints for internal users to fetch issue data or trigger sync operations, delivered via CloudFront and Route 53 for low latency.
5. **Monitoring & Logging** – CloudWatch captures all logs/metrics, and the team can review these or get alerted on anomalies.

This stack uses managed services to meet the requirements with minimal infrastructure management, aligning well with the company's internal use and rapid development goals.

Implementation Approach

We will implement the system in Python, following an event-driven pipeline for JIRA events and a modular design for the API. The development will be test-driven (to achieve 100% test coverage) and use continuous integration. Below is the step-by-step workflow and components of the implementation:

1. **JIRA Webhook Configuration:** We will configure JIRA to send webhooks for relevant events. For example, whenever an issue is created or updated (including status changes, comments, etc.), JIRA will send an HTTP POST to our API's endpoint (e.g. `https://jira-sync.internal.company.com/webhook`). This webhook includes details of the issue in XML format (as JIRA might send XML by default). We will ensure the webhook includes all necessary fields (JIRA allows selecting events or specifying the data format, if possible). The webhook will be set to fire *synchronously* so we get the data in near real-time.
2. **Webhook Ingestion Lambda (FastAPI):** The incoming webhook hits CloudFront, which invokes the **Webhook Handler Lambda**. This Lambda runs a FastAPI application to handle the `/webhook` route. FastAPI will parse the request and pass the XML payload to our handler code. The Lambda will immediately transform the **XML into JSON** – using an XML parser library to convert the structure into a JSON dict. We'll define a schema for the JSON (possibly matching JIRA's REST API JSON format for issues, for consistency). This transformation is fast and ensures subsequent steps deal with JSON (which is easier to work with in Python and store). The FastAPI route will quickly return a 200 OK response to JIRA (acknowledging receipt), so JIRA isn't kept waiting too long. Most processing can happen asynchronously after acknowledging, if needed.
3. **Data Processing & Storage:** After converting to JSON, the Lambda function will proceed to store and commit the data:
4. **Store to S3:** The Lambda uses the AWS SDK (boto3) to put the JSON object into the S3 bucket. The object key could be the JIRA issue key (e.g. `PROJECT-123.json`) for the latest state, and we might also add a time-based key for the specific event (if keeping a log of changes). For example, we could have `PROJECT-123/2025-02-09T19-00-00Z.json` for an event timestamped now. This ensures **all changes are captured** in S3 (either via versioning or separate files). The S3 write is straightforward and should succeed within milliseconds. (We will handle errors – e.g. if the put fails, the Lambda will log an error and possibly retry or send to a DLQ for manual intervention, to not lose data.)
5. **Update GitHub Repo:** Next, the Lambda will update the Git repository. Using the Git binary in the container, the Lambda will:
    a. Clone the repository (or fetch latest changes) into temporary storage (Lambda has /tmp space).
    b. Update or create the JSON file corresponding to the issue. For example, put the JSON into `data/PROJECT-123.json` in the repo.
    c. Commit the change with a message like "Update PROJECT-123 (JIRA webhook event at 2025-02-09T19:00:00Z)".
    d. Push the commit to GitHub (using the token from Secrets Manager for authentication). We might push directly to the main branch since this is an automated system, or optionally open a pull request if manual review is desired (likely not needed for internal data sync).

    e. If the push fails due to a race condition (e.g. another concurrent Lambda invocation pushed a change to the repo first), our code will catch the error, perform a `git pull` to reconcile, then retry the commit. This ensures that even if multiple issue changes come in at once, all will eventually be committed (some may require a couple of retries). We will serialize access as much as possible – e.g., we might use issue key as a scope for locking (two changes to the same issue in rapid succession could be combined or handled sequentially to avoid conflicts). Using GitHub as a storage provides a full history of changes (via commit history) and an extra backup. It also allows developers to use familiar tools (git diff, blame, etc.) to inspect how an issue changed over time.

6. **Continuous Integration & Testing:** All code for the Lambdas (and infrastructure definitions) will live in a GitHub repository (separate from the data repo). We will set up **GitHub Actions** as a CI/CD pipeline. Every time we push changes to the code repository (or open a PR), the pipeline will run:
7. **Automated Tests:** The test suite (written with unittest/pytest) will run, covering 100% of code paths. We will create unit tests for XML->JSON transformation (with sample webhook payloads), tests for the Git commit function (perhaps using a dummy repo or mocking Git calls), and tests for the FastAPI endpoints (using FastAPI's test client to simulate requests). Achieving *100% test coverage* means every function and edge case is tested – this is a project requirement to ensure reliability for this internal tool.
8. **Build and Deployment:** If tests pass on the main branch, the pipeline will proceed to build the deployment artifacts. Specifically, we'll build the Docker images for the Lambdas. One image will contain the webhook/Git handling code (with Git installed), and another image for the API endpoints (FastAPI code). These images are built using a Dockerfile (starting from an AWS Lambda Python base image and adding our code). After build, the pipeline will push these images to AWS ECR. Then, using AWS CLI or CloudFormation scripts, it will update the AWS Lambda functions to use the new image versions. We might use AWS SAM or CDK to define the infrastructure as code – in that case, the GitHub Actions can run `sam deploy` or `cdk deploy` to update the stack (which includes the Lambda functions, config, etc.). This CI/CD process means any code change is automatically tested and deployed with minimal human intervention, ensuring **continuous delivery** of improvements.
9. **FastAPI API Endpoints:** In parallel to webhook processing, we will implement additional FastAPI endpoints to allow internal users to retrieve and manage the synchronized data. This FastAPI application can be deployed on Lambda (behind Cloud Front) just like the webhook, possibly even the same FastAPI app could include the webhook route and other routes. For clarity, we might separate concerns (one Lambda specifically triggered by CloudFront for webhooks, and another for user API calls), or combine them if performance is not an issue. Key API endpoints planned:
10. `GET /issues/{issueKey}` – Retrieves the latest JSON snapshot of the specified issue from S3 (or optionally from the GitHub repo). This allows an internal tool or user to quickly fetch the current state of an issue without hitting JIRA directly.

11. `GET /projects/{projectKey}/issues` – Lists issues (or issue keys) for a given project that have been recorded. This can be achieved by listing objects in S3 with that prefix or maintaining an index. It provides a way to discover what data is available.
12. `GET /issues/{issueKey}/history` – (Optional) Returns a list of changes or links to past snapshots for that issue. This could pull commit history from GitHub or versions from S3 (if bucket versioning or multiple files are used). This shows the timeline of changes for an issue.
13. `POST /sync` – Triggers a manual sync job. This endpoint would be protected (perhaps only allow if a special token is provided) and, when called, would initiate a process to reconcile data with JIRA. For example, it could trigger a Lambda (or Step Function workflow) that calls JIRA REST API to fetch all issues and ensures the S3 and GitHub data matches (useful if we suspect a missed webhook or need a full backfill). This is a **Phase II** feature as initially we rely on webhooks, but having a manual full-sync option is good for completeness.
14. `GET /docs` and `GET /openapi.json` – These are provided by FastAPI automatically. The **Swagger UI** at `/docs` will allow users to explore the API and test calls, and the OpenAPI spec can be downloaded from `/openapi.json` for integration with other tools. We will customize the OpenAPI metadata (title, description, version) to clearly describe this internal API.
15. (Optional) **Admin endpoints**: If needed, we could add endpoints to manage configuration, e.g., `POST /webhooks` to programmatically register new webhooks in JIRA (if JIRA's API allows that), or endpoints to view system health (like `/health` returning status of connections to S3, GitHub, etc.). Initially, these might not be necessary or can be handled via AWS monitoring, but they are considered for future enhancement.

The API will use **JSON** payloads and responses throughout. Authentication can be handled by FastAPI (for instance, using a custom authorizer or API keys for internal clients), or since it's internal, we might simply restrict it at the network level (only accessible from the corporate network or VPN). We will start with basic security (at least an API key or token in requests) and plan to integrate with the company SSO or IAM in Phase II.

1. **Performance and Concurrency:** We will optimize the Lambda functions to handle the load. Each JIRA event is processed independently; AWS Lambda can scale out horizontally, so multiple events can be processed in parallel if needed. The processing itself (XML to JSON, S3 put, Git commit) is lightweight, typically completing in under a second or two. We'll allocate sufficient memory to the Lambda – note that Lambda CPU is tied to memory, so giving, say, 512MB or 1GB might significantly speed up the Git operations. We aim for end-to-end latency of only a few seconds from JIRA event to data saved. If certain operations (like Git) prove to be a bottleneck under high concurrency, we might introduce a queue (AWS SQS) to buffer and serialize them, but our initial design attempts a direct approach for simplicity. We will also use CloudWatch to monitor execution time; if we see it creeping up or any timeouts, we will refactor accordingly (e.g., move heavy work to an async background step). The system should be able to handle bursts (Lambda has a high concurrency limit by default, e.g., 1000 concurrent executions, which is plenty for our use case).

2. **Error Handling & Logging:** Throughout the implementation, careful error handling will be in place. If the webhook Lambda fails processing for some reason (e.g., unable to reach GitHub or S3), it can catch the exception and either retry or send the event to a Dead Letter Queue (AWS Lambda DLQ/SQS) for later reprocessing. We will log all failures with details to CloudWatch. Additionally, we can use AWS CloudWatch Alarms to notify the team if repeated errors occur (for example, if 5 webhook processing failures occur in a row, send an alert to email/Slack). This ensures the team can respond quickly to any issues (like a credential expiring or an outage in one of the dependencies).

Overall, the implementation approach emphasizes **automation** and **speed**: automated triggers via webhooks, automated testing and deployment via CI/CD, and automated recovery from errors where possible. By structuring the code with FastAPI, we also get a clear organization of routes and background tasks, making the system easier to extend later.

API Documentation & Design

We will design a clear and well-documented REST API as part of this system, primarily for internal developers/teams who want to query the JIRA data or trigger certain actions. Using **FastAPI** as the framework gives us automatic documentation generation. Key aspects of the API design:

- **Base URL & Versioning:** The API will be served at a base URL (for example, `https://jira-sync.internal.company.com/api/v1`). We include a version in the path (`v1`) to allow future changes without breaking clients. The Route 53 DNS and CloudFront will ensure this domain is accessible internally. Under the hood, CloudFront will route requests to the FastAPI Lambda.
- **OpenAPI Schema:** FastAPI will generate an OpenAPI schema documenting all endpoints, their request/response models, and any error responses. We will customize the schema with proper descriptions. The documentation will be accessible via **Swagger UI** and **ReDoc** (FastAPI provides `/docs` and `/redoc` by default). This interactive doc is crucial for internal users to understand how to use the API. Since this is internal, we can expose the docs openly (or protect with a simple auth if needed).
- **Endpoints and Models:** We will define Pydantic models in FastAPI for the data we return, matching the JIRA issue structure (e.g., fields like key, summary, status, etc.). Some core endpoints:
- `GET /api/v1/issues/{issueKey}` – Returns the latest issue data in JSON. On success: HTTP 200 with a JSON body containing all fields of the issue (as captured). If the issue is not found in our data store, return 404. We'll document the JSON structure in the OpenAPI schema (which will

reflect our Pydantic model).

- `GET /api/v1/projects/{projectKey}/issues` – Returns a list of issue keys (or brief info) for all issues in that project that we have stored. Could support query params for filtering. Useful for enumerating data.
- `GET /api/v1/issues/{issueKey}/history` – Returns a list of timestamped changes for that issue. This could be an array of objects each containing a timestamp and either the full snapshot or a diff of fields changed. If implementing fully might be complex, we may initially leave this endpoint for Phase II, but design the API with this in mind.
- `POST /api/v1/sync` – Kicks off a full sync as described earlier. The request could allow specifying a project or all projects. The response would likely be 202 Accepted with a task ID (if we run an async job) or 200 if it's quick. This is an administrative endpoint and will be secured (maybe require a special token or admin privileges).
- `GET /api/v1/health` – A simple health-check endpoint that returns status (could check connectivity to S3, GitHub, etc.). This helps in monitoring and can be used by an uptime check.
- **Security & Auth:** Initially, since this is for internal use, we may rely on network security (only accessible within our VPN or corporate network). However, if needed, we can issue an API key that must be included in requests (FastAPI can check a header for a key). In Phase II, we might integrate with our internal SSO (e.g., using OAuth tokens or JWTs) to authenticate users calling the API, especially if we open it up to more users. All endpoints will require HTTPS (which is ensured by CloudFront with our TLS certificate).
- **Rate Limiting and Quotas:** Given the internal nature and presumably moderate use, we might not need strict rate limiting. But CloudFront can enforce a throttle (e.g., 100 requests per second) to prevent abuse or bugs from spamming the system. We will document any such limits in the API docs.
- **Response Formats:** All responses will be in JSON format. Errors will follow a consistent structure (FastAPI by default might return validation errors in a structured way). We'll document possible HTTP status codes for each endpoint (e.g., 200 for success, 404 for not found, 500 for server error, etc.).
- **Example Usage:** The documentation will include example requests and responses. For instance, the Swagger UI will allow users to try out `GET /issues/{issueKey}` and see a sample response. We will ensure that our API is intuitive – using plural nouns for collections (`issues`) and singular for single resources, etc., consistent with REST guidelines.

- **Testing the API:** As part of the project, we will also write integration tests for the API endpoints. We can use the FastAPI provided test client to simulate calls in our CI pipeline to ensure the endpoints behave as expected.

By using **FastAPI**, we leverage a modern, high-performance framework that natively supports asynchronous I/O (useful if we ever need to call external APIs within requests) and integrates well with Pydantic for data validation. The OpenAPI documentation generated will be crucial for adoption of this internal tool, reducing the need for separate documentation writing. We will host the documentation (Swagger UI) such that any internal developer can easily access it via the web browser (with appropriate access control).

Overall, the API design is **RESTful, intuitive, and documented**. It provides not just a data dump, but a convenient way to query and monitor JIRA data via familiar HTTP calls, which can be used in scripts, other internal applications, or data analysis pipelines.

Financial Analysis

One of the advantages of a serverless architecture is that costs scale with usage and are minimal for low to moderate workloads. We have analyzed the expected AWS costs for this system to ensure it remains cost-effective for internal use. Below is a breakdown of the primary cost components:

- **AWS Lambda**: Lambda charges based on the number of invocations and the duration (GB-seconds) used. The AWS Lambda free tier provides *1 million free requests per month* and *400,000 GB-seconds of compute time per month* (this free tier is continuous, not just 12 months) (AWS Lambda Price Explained (With Examples) | Dashbird). Beyond that, the cost is $0.20 per 1 million requests and ~$0.00001667 per GB-second of execution (Amazon S3 Pricing - Cloud Object Storage - AWS). In our scenario, each JIRA event triggers one Lambda execution. Even in a busy month with, say, 100,000 issue updates, we'd be far below the free 1 million requests. The processing time per event might be around 100-200ms with 512MB memory (0.1-0.2 * 0.5 GB-s = 0.05-0.1 GB-s per exec). So 100k events would use ~5,000-10,000 GB-s, also within the free tier. In summary, **Lambda costs are effectively $0** at our scale, or at most a few cents if we go beyond free tier. Even if we had a million events, the cost is about $0.20 for requests + maybe ~$0.17 for compute – roughly **$0.37** for a million events, which is extremely low (Amazon S3 Pricing - Cloud Object Storage - AWS). We will monitor concurrency and if we need to raise limits (AWS default concurrency limit is 1,000, which is free to increase if needed).
- **Amazon S3**: S3 costs include storage and requests. **Storage**: S3 Standard storage is about **$0.023 per GB-month** for the first 50 TB (The No BS Guide To Understanding S3 Storage Costs - CloudZero). Our data consists of JSON text, which is quite small per issue. For example, if we track

10,000 issues with an average of 5KB each in JSON, that's ~50 MB of data. Even including historical versions, let's say 10 versions each, that's 500 MB. At $0.023/GB, 0.5 GB costs around **$0.0115 per month** – basically a penny. Even scaling up to hundreds of thousands of issues, the storage cost would be just a few dollars per month. **Requests**: We will have PUT (or POST) requests for each webhook event writing to S3, and GET requests when the API or processes read from S3. S3 pricing per 1,000 requests is roughly $0.005 per 1,000 for PUT/LIST and $0.0004 per 1,000 for GET (Amazon S3 Pricing - Cloud Object Storage - AWS). So, 100k events -> 100k PUTs = 100 * $0.005 = $0.50. GET requests by the API might also be on the order of thousands, incurring only pennies. Additionally, new AWS customers get 5 GB storage, 20k GET, and 2k PUT requests free per month (Amazon S3 Pricing - Cloud Object Storage - AWS). In our case, after development, this likely won't matter much, but it's nice to note that small usage is often covered by free tier. Overall, **S3 cost will be very low**, likely under $1/month in practice, unless we scale to millions of issues.

- **Amazon CloudFront**: CloudFront costs for serving our API and content include data transfer out and request fees. Data transfer out from CloudFront to end-users (our internal users) is $0.085 per GB for the first 10 TB per month in North America/Europe regions (Understanding CloudFront Request and Data Transfer Costs). Our use case will have tiny data payloads (JSON responses, docs, etc.), maybe a few MBs per day at most, which is far less than even 1 GB per month. So data transfer via CloudFront will cost only a few cents (if not effectively $0). CloudFront also has a free allocation of 1 TB for new sign-ups for a year, which we won't exceed initially. **Request fees**: $0.01 per 10,000 HTTPS requests (Understanding CloudFront Request and Data Transfer Costs) (about $1 per million). If we serve 100k requests in a month, that's 10 cents. So CloudFront might add cents to a dollar per month range. We budget perhaps **<$1/month** for CloudFront under typical usage. The benefit it gives (caching, performance) is well worth this negligible cost.
- **Amazon Route 53**: Route 53 hosted zone costs $0.50 per month for the domain (since we'll likely use one hosted zone for our internal domain). DNS query costs are $0.40 per million queries for the first billion (Amazon Route 53 pricing - Amazon Web Services). Internal services will do some DNS lookups, but this is typically extremely low (and often cached in clients). So Route 53 costs will be about **$0.50/month** fixed, with maybe a few cents for queries (if any significant number of lookups happen). If we create additional subdomains or records, those are free up to 10k records per zone (we'll have only a handful).
- **AWS Secrets Manager**: As noted, Secrets Manager charges $0.40 per secret per month (How to rotate a WordPress MySQL database secret using AWS Secrets Manager in Amazon EKS | AWS Security Blog). We anticipate storing possibly 2-3 secrets (GitHub token, maybe JIRA API token, maybe a general config secret) – so roughly **$0.80 to $1.20 per month**. API calls to retrieve secrets (during Lambda init) are $0.05 per 10,000 calls (How to rotate a WordPress MySQL database secret using AWS Secrets Manager in Amazon EKS | AWS Security Blog); our Lambdas will fetch secrets likely once per cold start. Cold starts might be a few per day, so practically $0 in API call charges.

- **CloudWatch Logs and Monitoring**: CloudWatch Logs costs $0.50 per GB ingested and $0.03 per GB archived per month. Our logs will be mainly text logs from Lambdas. We will keep logging concise (not dumping large payloads unnecessarily). Suppose each event logs 1KB of data and we have 100k events = 100k KB = ~0.1 GB of logs per month, that's $0.05. Even with verbose logging, it will be well under **$1/month**. Metrics (default ones) are free, and even custom metrics are cheap ($0.30 per metric/month) if we add any. We might set up a few CloudWatch Alarms ($0.10/alarm/month), maybe 2-3 alarms, so that's <$1. So CloudWatch total might be on the order of **$1-2/month** at most, likely less.
- **GitHub Costs**: Using GitHub itself does not incur AWS charges, but we consider if there are any costs on GitHub's side:
- The GitHub repository where JSON is stored will grow in size as we accumulate history. However, text JSON compresses well in git and even a large history is likely only tens of MBs. This is well within GitHub's repo size limits (and no cost unless we are storing huge files or using LFS).
- GitHub Actions CI/CD minutes – If using a public repo, Actions are free; for private, we get a certain amount of minutes free (e.g., 2,000 Linux minutes per month on a typical plan). Our CI runs (tests + build) might take ~5-10 minutes per deploy. If we deploy say 20 times a month, that's 200 minutes, within free limits. If on GitHub Enterprise Cloud, we might have an allotment or it's a corporate account – anyway, CI cost is not likely an issue. We will monitor usage.
- There is no direct cost per API call or push to GitHub. So integration is free aside from the development effort.
- **Other AWS Services**: IAM has no direct cost. ECR has minimal cost (first 500MB storage is free, and our images likely under that; data transfer within same region for Lambda pulling image is free). If we use Step Functions for the `/sync` full sync in future, that would cost a bit ($0.025 per 1,000 state transitions), but that's Phase II perhaps.

In summary, under expected usage, the **monthly AWS cost** of this system can be on the order of **a few dollars or less**. Most of the serverless services (Lambda, S3, CloudFront) will likely fall in the free tier or only marginally above it for our scale.

For the value it provides (real-time backups and an accessible API for all JIRA changes), this cost is quite reasonable. Additionally, the cost scales with usage: even if our JIRA usage doubles, the costs would only double in proportion (still small). We will also set up AWS Cost Alerts to monitor if costs unexpectedly jump (e.g., due to a bug causing a flood of Lambda invocations or a misconfiguration). This financial plan ensures the project stays **within budget** and demonstrates high cost efficiency thanks to the serverless design (AWS Lambda Price Explained (With Examples) | Dashbird) (Amazon S3 Pricing - Cloud Object Storage - AWS).

Deployment

Deployment Process: We will use an automated deployment pipeline to release this system into AWS. Our approach is to integrate deployment with the GitHub Actions CI workflow: - For infrastructure, we will define everything as code (using AWS SAM or AWS CDK). This includes Lambda functions, S3 bucket, IAM roles, etc. These definitions live in the repository. - When changes are pushed to the main branch (after passing tests), the GitHub Actions workflow will package and deploy the new code. For example, using AWS SAM: `sam build` to package code and Docker images, `sam deploy --no-confirm-changeset` to apply changes. Or with CDK: `cdk synth` and `cdk deploy`. We will configure GitHub Actions with AWS credentials (stored securely in GitHub secrets) that have permission to deploy the stack (likely an IAM user or role with CloudFormation deploy rights). - The deployment will cover updating Lambda code (pointing to the new Docker image digest in ECR or new function code) and any infrastructure changes. Because it's serverless, deployments are quick and have minimal downtime. Lambda updates are atomic (new version deployed and then traffic switched). - We will maintain separate environments if needed: e.g., a "dev" stage and a "prod" stage. This could be separate AWS stacks or even separate AWS accounts. The CI pipeline can deploy to a dev environment on each commit and require a manual approval to deploy to prod. This ensures we can test changes in a staging area with minimal risk. - **DNS and CDN Deployment:** We will use Infrastructure as Code to also set up Route 53 records and CloudFront distribution. For example, a CloudFormation template can create the CloudFront distribution with the proper domain name and attach it to Route 53. One consideration: provisioning an SSL certificate via AWS Certificate Manager for our domain (e.g., `*.internal.company.com`) – we need to do that and ensure CloudFront use it. This will be part of the initial setup. Subsequent deployments will reference the existing certificate.

In conclusion, the deployment strategy ensures that launching and updating the system is **automated and safe**, using CI/CD best practices. Once deployed, we will document operational runbooks (e.g., how to redeploy if something goes wrong, how to rotate the GitHub token with minimal downtime, etc.) so the system can be maintained by the team with confidence.

This project plan provides a comprehensive approach to delivering a serverless JIRA exporter & sync system that meets the requirements. It leverages AWS services to minimize upkeep and maximize scalability, integrates with development workflows (GitHub) for efficiency, and lays out a clear path for implementation, testing, and future growth. With this plan, the team can proceed to implementation knowing the objectives, architecture, and steps to success. The result will be a reliable internal service that keeps a near-real-time backup of JIRA data and makes it accessible and auditable to our internal stakeholders, all while keeping costs and maintenance burden very low.