# Kaiser Permanente 🡪

**🔍 Overview**

This project involved modernizing a legacy monolithic healthcare backend application into two microservices. Application provides backend services to internal applications that interacted through REST and SOAP APIs via an API Gateway. The services were responsible for managing patient data and synchronizing demographic information across systems, including external third-party databases (**EPIC** widely used **Electronic Health Record (EHR)** systems acts as a **single source of truth** for patient records. It stores detailed patient demographics, medical history, treatments, lab results, medications, and insurance details). The main objective was to improve modularity, maintainability, scalability, and observability and fault isolation while supporting both REST and SOAP integrations.

**🧱 Architecture and Components**

**🧩 Microservices Breakdown:**

1. **Patient Service (Microservice 1):**
   * Responsible for setting up, updating, and retrieving general patient information (e.g., account details).
   * Validates incoming requests (e.g., null checks and required fields).
   * Interacts with third-party systems via SOAP to retrieve and update patient data.
2. **Demographics Service (Microservice 2):**
   * Handles patient demographic details exclusively.
   * Updates local records and synchronizes with a third-party system for consistency.
   * Also, can receives internal REST API calls from Microservice 1 to update demographic data.

**🔄 Service Interaction Flow:**

* A request enters through the API Gateway and reaches Microservice 1.
* After request’s validation, it makes a SOAP call to a third-party system to fetch current data.
* Combines the new and existing data, then makes a SOAP call again to update the third-party system.
* Call Microservice 2 internally via REST to update demographic info.
* Returns an appropriate response to the caller.

**🧰 Technologies & Why They Were Used:**

| **Tech/Tool** | **Purpose** | | **Why It Was Used** |
| --- | --- | --- | --- |
| **Spring Boot** | Backend development | | Lightweight, fast setup for microservices, integrates well with REST/SOAP |
| **Spring Config Server** | Externalized configuration | | Centralized management of configuration across services; environment flexibility |
| **Microservices Architecture** | Splitting monolith into manageable services | | Scalability, modularity, independent deployments |
| **Docker** | Containerization | | Ensures consistency across development and deployment environments |
| **Azure Kubernetes Service (AKS)** | Deployment | | Managed Kubernetes for scaling, service discovery, and deployment orchestration |
| **API Gateway** | Routing & abstraction | | Central point to manage traffic, secure endpoints, expose REST/SOAP |
| **SOAP & REST** | Inter-service and third-party communication | | Needed due to legacy third-party systems (SOAP) and modern internal services (REST) |
|  |  |  | |
| **JUnit** | Unit Testing | | Ensures code quality and catches bugs during development |
| **Splunk** | Centralized log aggregation and monitoring | | Real-time log analysis and debugging. Search, visualize, and analyze logs in real time using correlation ID |
| **Fault Tolerance (Resilience4j/Hystrix)** | Improve resiliency | | Prevents cascading failures when external systems are slow/unavailable |
| **Circuit Breaker** | Service protection | | Temporarily halts requests to failing services to maintain system health |
| **Correlation ID** | End-to-end traceability across microservices | | Helps in debugging and tracing request flow |
| **Smoke Tests** | Health validation | | Quick checks after deployment to ensure critical endpoints are functional |

**👨‍💻 Your Role & Contributions:**

* Contributed to backend development and service refactoring from monolith to microservices.
* Worked on POCs for spring cloud configurations. Soap calls.
* Worked on integrating REST and SOAP APIs through the API Gateway.
* Implemented request validation logic (null checks, required fields).
* Integrated logging using a consistent format for easy traceability.
* Configured Spring Boot to fetch environment-specific configs from a central config server.
* Wrote JUnit tests to ensure correctness of business logic and validations.
* Gained familiarity with Splunk for observing logs during debugging and issue resolution.
* End to end testing of functionality and bug fixes if any.

**🧩 Challenges Faced & How They Were Solved**

**1. 🔄 Monolith to Microservices Migration Complexity**

* **Challenge:** Migrating a monolithic application to microservices architecture without disrupting existing workflows or downstream systems.

One major challenge was breaking down the monolith. We had to be really careful to avoid breaking existing flows. We identified logical boundaries — one service for general patient data and another for demographics. We made sure the API contracts between them were clean, and that external consumers didn’t notice any disruption.

**2. 🧪 Ensuring Request Validity and Data Integrity**

* **Challenge:** Incoming requests were prone to incomplete or malformed data, especially from legacy clients.

Since we had other internal systems calling our services, sometimes the requests were incomplete — missing required fields or malformed data. I handled this by writing backend validations — mostly null checks, required fields, and format validations — before any processing began. That reduced a lot of silent failures downstream.

**3. ☁️ Centralized Configuration Management Across Environments**

* **Challenge:** Managing service-specific configurations (like URLs, timeouts, third-party credentials) across multiple environments (dev, staging, prod) was complex and error-prone.

Each service needed environment-specific configs — like third-party URLs, timeouts, etc. Managing those in YAMLs or hardcoded values wasn't sustainable. I integrated Spring Cloud Config Server so all configs could be externalized and dynamically updated without redeployments.

**4. 🧱 Legacy System Integration via SOAP**

* **Challenge:** The third-party system supported only SOAP, making it harder to integrate with modern REST-based services.

A tough one was the SOAP integration with a third-party system. It only accepted and returned SOAP XML, which was a challenge since we used REST internally. The team built SOAP request templates and handlers to convert REST payloads to SOAP envelopes and back. It was tricky, but worked reliably.

**5. 🔥 Lack of Fault Tolerance in Early Stages**

* **Challenge:** Initial service-to-service and external calls were not resilient. A failure in the third-party system caused cascading failures in the app.

Initially, when the third-party system was down, our entire app would hang or fail. To fix this, my teammates implemented circuit breakers and retry logic so we could handle failures gracefully instead of crashing the service.

**6. 🔍 Debugging & Observability**

* **Challenge:** It was difficult to trace issues across multiple services without centralized logs or correlation IDs.

With multiple services and external systems involved, it was hard to trace issues. The team added correlation IDs to log each request end-to-end, and integrated with Splunk for centralized logging. I contributed to setting up logging in the services I worked on.

**7. 🚀 Post-Deployment Confidence**

* **Challenge:** Manual testing wasn't scalable after deployments, leading to uncertainty in service stability.

After deployment, we needed a way to quickly verify the service was up and healthy. My teammates wrote smoke tests that hit critical endpoints automatically. It gave the team confidence that the service was behaving as expected in prod.

**🌟 Business Value Delivered**

* Improved **scalability** and **modular development**
* Enhanced **observability** and **fault isolation**
* Enabled **smoother integrations** with internal and external systems
* Reduced **deployment risk** through containerization and Kubernetes orchestration

**🏥 Resume Project: Healthcare Backend Modernization**

*Duration: 1.5 Years |* Java, Spring Boot, REST/SOAP, Docker, Kubernetes (AKS), Splunk, Azure, API Gateway

**Healthcare Backend Modernization**  
*Role: Backend Developer | Java, Spring Boot, REST, SOAP, Docker, Kubernetes (AKS)*

* Contributed to migrating a legacy monolithic healthcare app into two spring boot microservices to improve scalability and modularity and maintainability.
* Designed and developed java backend APIs for patient registration, enabling CRUD operations on account, attendee, demographic, and medical data.
* Built RESTful APIs and SOAP clients to communicate with internal services and legacy third-party systems.
* Wrote unit and integration tests using JUnit and Mockito to ensure code quality and reliability and maintain sonar code quality to support CI/CD readiness.
* Collaborated with QA and DevOps for testing end-to-end functionality, trace logs via Splunk for real-time issue analysis, and resolve bugs post-deployment in Azure K8.
* Connected services to Spring Cloud Config Server and a global config server to manage dynamic, environment-specific configurations across environments centrally from Git.
* Worked with Ignite API Gateway (Apigee) to route, aggregate, and secure APIs as a reverse proxy across services.
* Followed Agile methodology and participated in full product development lifecycle including development, testing, and post-deployment support.

**🧠 General Understanding**

**💬 Q: Why did you break the monolith into microservices?**

To improve scalability, maintainability, and deployment flexibility. The monolith was tightly coupled, so even small changes required full redeployment. By splitting it, we isolated patient account and demographics handling, making the system easier to manage and scale independently.

**💬 Q: How did you break the monolith into microservices?**

We identified logical service boundaries — one for general patient info and another for demographics. We refactored the codebase, ensured clear API contracts, and established communication between services via REST. This gradual extraction helped preserve functionality.

**💬 Q: Can you explain the communication between your two services?**

The first service handles requests, performs validations, and then internally calls the second service to update or retrieve demographic data. It then aggregates the response and sends it back to the client. We used REST-based internal service communication.

**💬 Q: How did you ensure backward compatibility with the legacy SOAP systems?**

We built SOAP clients that handled XML request/response conversions, keeping the interface to third-party systems unchanged. Our microservice abstracted away the internal REST changes, so external systems could continue calling via SOAP as before.

**💬 Q: How did you move application from JDK 1.2 to JDK 8? Why not the latest version?**

JDK 8 was selected due to compatibility and stability with our existing libraries. The team prioritized a stable upgrade path rather than pushing to the latest version, which would have required testing all dependencies for compatibility.

**🔧 Your Contributions**

**💬 Q: What kind of validations did you implement and why?**

I handled backend-level validations like null checks, required field validations, and format validations to ensure clean, safe data before any processing or downstream calls. This reduced errors and improved data integrity.

**💬 Q: How did you structure your JUnit tests?**

I used a modular structure — individual test classes per service or controller. Tests were written using JUnit and Mockito, covering both positive and negative cases, with mock data for isolated unit testing.

**💬 Q: How did you configure your service to pull values from the Spring Config Server?**

I added Spring Cloud Config dependencies and configured the bootstrap.yml to point to the Git-backed config server. Services would fetch values dynamically at startup based on the active profile/environment.

**🏗️ Architecture & Resiliency**

**💬 Q: What fault tolerance mechanisms were used?**

Circuit breakers, retries, and timeouts. These helped in cases when the third-party or internal services were slow or down, ensuring the system didn't become unresponsive.

**💬 Q: What’s the role of the circuit breaker pattern in your system? How did you use it?**

Circuit breakers prevent repeated calls to a failing service. When failures hit a threshold, the breaker trips and stops further calls for a set time, allowing recovery. This avoided cascading failures across services.

**💬 Q: How are logs managed and monitored using Splunk?**

Services were configured to generate structured logs with correlation IDs. These logs were pushed to a centralized location monitored via Splunk, where we could search logs by request ID, error types, and timestamps for faster debugging.

**💬 Q: What is a correlation ID, and how is it used in your system?**

A correlation ID is a unique identifier added to each incoming request. It’s passed through all downstream services and included in logs, enabling us to trace an entire request flow across services for debugging and monitoring.

**🚀 DevOps & Deployment**

**💬 Q: How is your service deployed?**

It’s packaged as a Docker image and deployed to Azure Kubernetes Service (AKS) using Helm charts. CI/CD pipelines handle the build and deploy stages.

**💬 Q: Why use Docker with Kubernetes for deployment?**

Docker ensures consistency across environments by packaging code with dependencies. Kubernetes provides auto-scaling, self-healing, and better resource management. Together, they offer reliable, scalable deployments.

**💬 Q: How do you handle config changes across environments?**

All config properties are stored in a Git-backed Spring Cloud Config Server. Each service reads environment-specific configs (Dev, QA, Prod) at runtime using Spring profiles.

**💬 Q: What are smoke tests, and how are they different from unit/integration tests?**

Smoke tests verify critical service functions post-deployment to ensure it’s running correctly. Unit tests check isolated functions, and integration tests check how modules work together — smoke tests come last to validate deployments.

**💬 Q: What is regression testing?**

It ensures that recent code changes haven’t broken existing functionality. It’s run as part of QA testing whenever new features are added or bugs are fixed.

**💬 Q: What kind of deployment was done in your project? Green deployment?**

We mostly used rolling deployments in AKS. Green-blue deployments were not explicitly used, but staging and production environments were maintained separately for safe rollouts.

**💬 Q1: Why exactly did you choose SOAP? When do you choose SOAP over REST?**

**Answer:** SOAP was chosen for communication with the third-party system because:

* The **third-party system only supported SOAP**.
* SOAP offers **strict contract enforcement** via WSDL (Web Services Description Language), which is crucial in regulated domains like **healthcare**.
* It provides **built-in features like WS-Security**, **message-level encryption**, and **ACID-compliant transactions**, which are often required in enterprise and legacy systems.

**When to choose SOAP:**

* You need **strong security standards**, **transaction support**, or **formal service contracts**.
* You're integrating with **legacy systems** that only expose SOAP APIs.
* You need **stateful operations**.

**When to choose REST:**

* You want **lightweight**, **stateless**, and **scalable** APIs.
* You’re working with **web/mobile applications**.
* You need **faster development** and **easier client consumption** (especially JSON).

**💬 Q2: How was the API Gateway helpful in your system?**

**Answer:** The API Gateway played a central role by:

* Acting as a **single entry point** for all external consumers.
* Handling **protocol translation** between REST and SOAP when needed.
* Performing **authentication, authorization**, **rate limiting**, and **request throttling**.
* **Routing** incoming requests to the appropriate microservice.
* Simplifying the client-side logic — consumers didn’t need to know about multiple internal endpoints.

**💬 Q3: How did authentication work in your application?**

**Answer:** Authentication was handled at the API Gateway level. Some common approaches (depending on the actual implementation) include:

* **OAuth 2.0 / JWT tokens**: Clients authenticate and receive a token, which is passed in each request.
* **API Keys or Certificates** for SOAP consumers.
* The Gateway validates these credentials before forwarding requests to downstream microservices.
* **Role-based access** might also be applied using claims inside the token.

**💬 Q4: How was the application secured overall?**

**Answer:** The application followed several security best practices:

* **Authentication and authorization** at the API Gateway.
* **SSL/TLS encryption** for all communication.
* **SOAP WS-Security** for third-party SOAP calls.
* **Secrets and configurations** were managed using Azure Key Vault or Kubernetes secrets.
* Regular **vulnerability scans**, dependency checks, and **limited role-based access to resources**.

**💬 Q5: How was the application deployed to Kubernetes?**

**Answer:**

* Each microservice was **containerized using Docker**.
* Docker images were stored in a container registry (e.g., Azure Container Registry).
* Deployment was done on **Azure Kubernetes Service (AKS)** using:
  + **Helm charts** or **YAML manifests** for defining deployments, services, and config maps.
  + **CI/CD pipelines** (e.g., Azure DevOps or GitHub Actions) for automated build and deployment.
  + Kubernetes handled **scaling, rolling updates**, and **self-healing**.

**💬 Q6: How does the application handle load?**

**Answer:**

* **Horizontal scaling** was enabled on Kubernetes (pods scale out based on CPU/memory usage).
* Kubernetes HPA (Horizontal Pod Autoscaler) can automatically increase instances under high load.
* **API Gateway rate limiting** protects services from abuse.
* **Caching** could be implemented on frequent reads (e.g., Redis).
* **Load balancers** distribute traffic across pods evenly.

**💬 Q7: What happens if one microservice fails? How is failure handled?**

**Answer:**

* **Circuit breakers** prevent cascading failures. If a microservice is down, the breaker opens and falls back to a default response or error handling logic.
* Kubernetes restarts failed pods automatically (self-healing).
* API Gateway can return a meaningful error if a backend service is unavailable.
* Services were designed to be **loosely coupled** — failure in one doesn’t necessarily bring down the whole system.
* **Logging and alerting** (via Splunk and monitoring tools like Prometheus/Grafana) notify the team immediately.

**🔥 Advanced Interview Questions & Answers**

**💬 Q1: How did you ensure backward compatibility while migrating from monolith to microservices?**

**Answer:** During migration, we followed a **strangling pattern** — exposing new microservices via the same API Gateway while keeping the monolith active. Gradually, functionalities were rerouted to the new services. We:

* Ensured that the **API contracts remained unchanged** (or versioned them when needed).
* Maintained **dual-write logic** temporarily when syncing data.
* Used **feature toggles** to test new services in production before fully switching over.

**💬 Q2: What was your approach to versioning APIs in your microservices?**

**Answer:** We used **URI versioning** (e.g., /api/v1/patient) for clear separation. This allowed:

* Parallel support of multiple versions (e.g., v1 and v2).
* Safe evolution of services without breaking existing consumers. We also ensured **semantic versioning** was used on internal shared libraries to control compatibility.

**💬 Q3: How do you handle data consistency across microservices?**

**Answer:** We aimed for **eventual consistency** where possible. To handle cross-service consistency:

* We used **synchronous APIs** for critical reads/writes.
* For other flows, **event-driven architecture** could be adopted using tools like Kafka (in other contexts).
* We also applied **idempotency** and **retry logic** to avoid duplicate updates or lost data.

**💬 Q4: What’s your strategy for handling long-running SOAP calls that can time out or slow down the system?**

**Answer:** To prevent timeouts and resource blocking:

* We used **asynchronous processing** for non-blocking flows (e.g., queuing background sync jobs).
* Implemented **timeouts and retries** with exponential backoff.
* Wrapped SOAP calls with a **circuit breaker (e.g., Resilience4j or Hystrix)** to avoid cascading failures.
* Logged failures with **correlation IDs** for traceability and alerts.

**💬 Q5: How do you monitor the health of your services in production?**

**Answer:**

* We exposed **health check endpoints** (e.g., /actuator/health in Spring Boot).
* Kubernetes used **liveness and readiness probes** to manage pod health.
* Logs were pushed to **Splunk** for analysis and alerting.
* We used (or could use) **Prometheus + Grafana** for metrics dashboards and custom alerts.
* Set up **alert rules** based on error rates, latency, and service unavailability.

**💬 Q6: How do you ensure secure communication between services in Kubernetes?**

**Answer:**

* Services used **mutual TLS (mTLS)** for secure service-to-service communication.
* Secrets (like certificates and tokens) were managed using **Kubernetes Secrets** or **Azure Key Vault**.
* API Gateway was the only publicly exposed component — internal services were **not internet-facing**.
* Enforced **network policies** to restrict access between namespaces or services.

**💬 Q7: What would be your approach to scaling this architecture if usage doubles or triples?**

**Answer:**

* **Horizontal scaling** via Kubernetes HPA.
* Use **distributed caching** (like Redis) to offload databases.
* Apply **database partitioning** or **replication** if DB becomes a bottleneck.
* Profile and **optimize API response times**.
* Revisit **circuit breaker thresholds and retry policies** under high load.

**💬 Q8: How did you contribute as a senior engineer?**

**Answer:**

* Led backend module development and ensured alignment with architecture goals.
* Mentored junior developers on REST/SOAP integration, containerization, and debugging.
* Reviewed PRs with focus on **readability, testability, and security**.
* Collaborated with DevOps team for CI/CD pipeline and Kubernetes deployment strategy.
* Participated in **design discussions**, providing alternatives and weighing trade-offs.

**💬 Q9: How would you debug a failed request that touches both microservices and the third-party SOAP API?**

**Answer:**

* Trace the **correlation ID** across all service logs in Splunk.
* Check the **API Gateway logs** for entry/exit points.
* Look at **circuit breaker status** (open, half-open, closed).
* Review **Kubernetes logs** for any pod restarts or crashes.
* If SOAP call failed, check **SOAP fault details**, response time, and any retry logs.

**💬 Q10: How do you test microservices that involve SOAP-based third-party systems?**

**Answer:**

* Use **mock servers** or tools like **SoapUI** to simulate third-party responses.
* Write **contract tests** to ensure we conform to WSDL specifications.
* Use **JUnit and integration testing** for SOAP request/response flows.
* Test fallback behavior by simulating third-party failures.

**🧠 More Senior-Level Interview Questions and Answers**

**💬 Q11: How do you handle logging in a distributed microservices architecture?**

**Answer:**

* We implemented **centralized logging** using **Splunk**.
* Each microservice logs structured messages with **context-rich metadata**, like:
  + correlationId
  + serviceName, operationName
  + userId or transactionId
* Logs are pushed via a log agent (e.g., Fluentd or Logstash) to Splunk.
* We built dashboards and alerts around **error rates, latencies**, and **service interaction traces**.

**💬 Q12: What are some risks of using synchronous communication between microservices? How did you mitigate them?**

**Answer:** **Risks:**

* Tight coupling between services.
* If one service fails, others may hang or fail too.
* Performance bottlenecks from chained calls.

**Mitigations:**

* Used **timeouts**, **retry policies**, and **circuit breakers**.
* Kept **payloads minimal** and used **idempotent endpoints**.
* Considered **asynchronous messaging or event-driven communication** for non-critical paths.

**💬 Q13: How did you manage configuration across environments (Dev, QA, Prod)?**

**Answer:**

* Stored environment-specific configs in **Kubernetes ConfigMaps** and **Secrets**.
* Used **Spring Cloud Config** or **Azure App Configuration** in some setups.
* Followed the **12-factor app** principles to externalize configs from code.
* Sensitive data (tokens, DB passwords) was encrypted and injected via **env variables or mounted secrets**.

**💬 Q14: Describe your CI/CD pipeline for this project.**

**Answer:**

* Code pushed to Git triggered a pipeline in **Azure DevOps**.
* Steps:
  + **Build the app**, run **unit and integration tests**.
  + **Build Docker images**, push to **Azure Container Registry**.
  + Apply **security scanning** (e.g., Trivy or Aqua).
  + Deploy to **AKS via Helm** or kubectl apply.
  + Notify Slack or email on success/failure.

**💬 Q15: How would you perform a blue-green or canary deployment in Kubernetes?**

**Answer:**

* For **blue-green**:
  + Maintain two deployments (v1, v2) behind a service.
  + Switch the service selector to the new version when ready.
* For **canary**:
  + Create a second deployment with the new version.
  + Use **Istio or Linkerd** to route a small % of traffic to it.
  + Gradually increase the traffic and monitor errors/performance.

**💬 Q16: How would you handle schema changes in a shared database between services?**

**Answer:**

* Use **backward-compatible changes** (add columns, don’t rename/drop).
* Implement **feature toggles** to delay usage until all consumers are ready.
* Use **database versioning tools** like Flyway or Liquibase.
* Introduce **read-models or data duplication** if shared DB becomes a bottleneck.

**💬 Q17: How would you secure a SOAP-based API integration with a third party?**

**Answer:**

* Used **WS-Security** with:
  + **UsernameToken**, **digital signatures**, **encryption**, or **mutual TLS**.
* Limited IP ranges/firewalls for inbound access.
* Logged and monitored all calls for anomaly detection.
* Implemented **replay attack prevention** using timestamps and nonces.

**💬 Q18: How do you make your microservices testable and maintainable?**

**Answer:**

* Followed **SOLID principles** and **clean architecture**.
* Wrote **unit tests**, **contract tests**, and **integration tests**.
* Mocked external systems using **WireMock**, **SoapUI**, or **MockServer**.
* Used **OpenAPI/Swagger** and **WSDL validation tools** for contract conformance.

**💬 Q19: How would you onboard a new developer to your microservice setup?**

**Answer:**

* Provide a **README with architecture, setup instructions**, and **dev scripts**.
* Give access to **Postman collections**, **OpenAPI specs**, and test credentials.
* Set up a local environment using **Docker Compose** or **Minikube**.
* Assign them **low-risk tickets** paired with code walkthroughs and PR reviews.

**💬 Q20: What trade-offs did you consider while designing your microservice architecture?**

**Answer:**

| **Area** | **Trade-off** |
| --- | --- |
| **Service granularity** | Fewer services = simpler ops; more services = better isolation |
| **Synchronous vs Async** | Sync = simplicity, but tighter coupling; async = resilience, but complexity |
| **Data ownership** | Duplicating data improves decoupling, but introduces consistency challenges |
| **Deployment speed** | Independent deployments are great but require strong CI/CD and observability |
| **SOAP vs REST** | Chose SOAP for compatibility and contracts, even though it’s heavier |

**🧨 What Happens If This Application Receives More Load?**

When your healthcare backend system receives significantly more load, **several components** are impacted. Here's how it breaks down:

**🔹 1. Increased Requests at the API Gateway**

* **What happens:**
  + The API Gateway starts receiving more incoming REST/SOAP calls.
  + If rate-limiting is configured, some requests might be throttled or rejected (429 errors).
* **How to mitigate:**
  + Auto-scale the gateway layer if it supports it (e.g., NGINX ingress or Azure API Management).
  + Apply **rate limiting** per client to avoid abuse.
  + Use a **global cache** (like Redis) for common, frequently accessed data.

**🔹 2. Microservices Under Load**

* **What happens:**
  + Increased CPU/memory usage → degraded response times → potential timeouts.
  + If one microservice is overwhelmed, it can affect dependent services (e.g., SOAP service slowdown affects the main patient service).
* **How to mitigate:**
  + **Horizontal Pod Autoscaling (HPA)** in Kubernetes to spin up more pods.
  + Set **resource limits/requests** in pod specs for optimal scheduling.
  + Use **circuit breakers** to prevent cascading failures (e.g., Resilience4j).
  + Apply **bulkheads** (isolation) to contain failures within a limited scope.

**🔹 3. Database Under Load**

* **What happens:**
  + More read/write operations can saturate DB connections, increase IOPS, or cause deadlocks.
* **How to mitigate:**
  + Implement **connection pooling** and limit max connections per service.
  + Use **read replicas** to scale reads.
  + Apply **caching** for non-volatile data (Redis, Memcached).
  + Analyze slow queries and add indexes or optimize schema.

**🔹 4. Third-Party SOAP System Bottleneck**

* **What happens:**
  + Third-party SOAP endpoints may not scale as well → become the bottleneck.
  + Timeouts or unresponsive services will affect the patient demographic service.
* **How to mitigate:**
  + Use **queue-based buffering** (e.g., store in SQS or Kafka, then process async).
  + Circuit breaker fallback with graceful degradation (e.g., return partial data or stale cache).
  + Adjust timeouts and retry logic intelligently.

**🔹 5. Logging and Observability Overload**

* **What happens:**
  + Log volume spikes → logging systems like Splunk may lag or drop logs.
* **How to mitigate:**
  + Use **log sampling** or rate-limiting.
  + Optimize log levels — e.g., avoid DEBUG logs in production.
  + Monitor key health indicators (latency, CPU, error rate) with **Prometheus/Grafana**.

**🔹 6. Deployment Resilience**

* **Mitigation techniques:**
  + Use **rolling updates** and **pod disruption budgets** to avoid downtime during scale-ups or updates.
  + Maintain **blue-green or canary deployment** setups for safe releases under high load.
  + Design services to be **stateless** and easily replaceable.

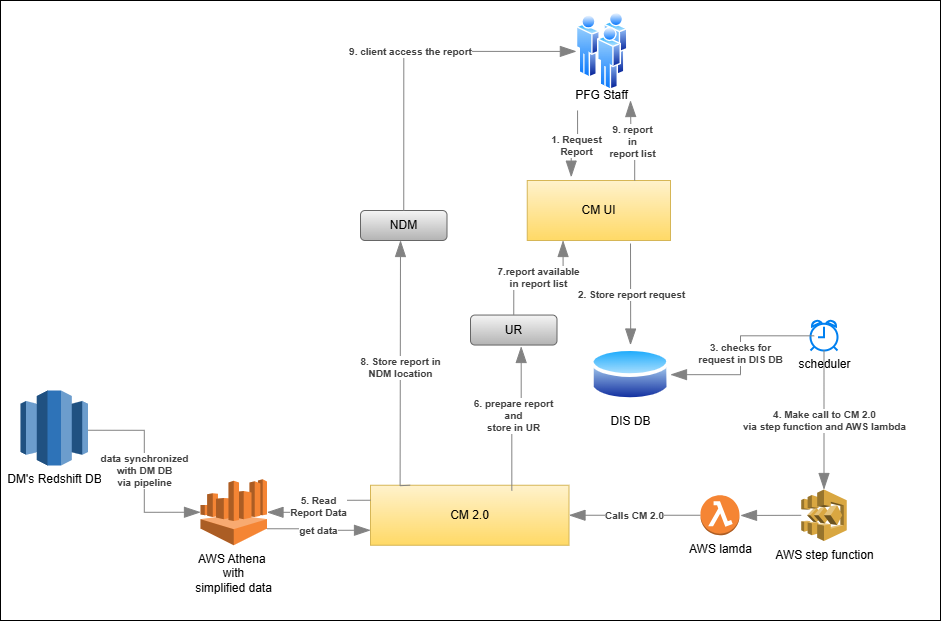
**✅ Summary: How You Show Readiness as a Senior Engineer**

In an interview, summarize your readiness like this:

"We designed the system with scalability in mind — Kubernetes handles auto-scaling, and each service is stateless and horizontally scalable. We've implemented circuit breakers and timeouts to isolate failures, used caching and read replicas to offload the database, and monitored system health via Prometheus and Splunk. If usage spikes, we're able to react gracefully without service disruption."

# Broadridge Project 🡪

## CM reporting🡪



**Application Flow: Report Generation System**

1. **EventBridge Rule** triggers the Lambda function every 5 minutes using a scheduled cron or rate expression (rate(5 minutes)).
2. **Lambda Function** queries **Athena** to retrieve the required data for the report.
3. Data retrieved from Athena is processed and mapped to a **custom report object**.
4. The generated report is stored in **Amazon S3** for persistent and secure storage.

**Optional Extensions (for Scalability and Optimization):**

* **Step Functions** can be added if more complex workflows are required after the Lambda function (e.g., sequential tasks, retries, or parallel processing).
* For larger datasets, you may consider **Athena partitioning** to improve query performance and reduce costs.

**Why AWS Lambda, Step Functions, EventBridge, and S3?**

**1. AWS Lambda:**

* **Purpose**: AWS Lambda is used for running code in response to triggers without managing servers. It's serverless, scalable, and cost-effective because you only pay for the compute time your code uses.
* **Why it’s used**: Lambda can be used to process reports on the fly (like generating reports or performing transformations on the data) without needing to provision or manage servers. The main benefit here is scalability—if the report generation is triggered by multiple events, Lambda automatically scales to handle them without requiring manual intervention.

**2. AWS Step Functions:**

* **Purpose**: AWS Step Functions is used to coordinate and orchestrate multiple AWS services into serverless workflows.
* **Why it’s used**: In your system, it helps to manage the workflow of report generation, including calling different Lambda functions, checking the data in S3, and ensuring that reports are generated in sequence. Step Functions helps in providing a visual interface for the workflow, retrying failed tasks, and managing complex tasks.

**3. Amazon EventBridge Scehdule:**

* **Purpose**: EventBridge is used for event-driven architecture, allowing you to trigger actions based on events from AWS services, integrated apps, or custom events.
* **Why it’s used**: It allows for seamless integration of asynchronous event-driven workflows. For example, when a report request is made, EventBridge can trigger the necessary Lambda function or Step Function, which processes the report generation. EventBridge enables decoupling, allowing for more scalable and manageable systems.

**4. Amazon S3:**

* **Purpose**: Amazon S3 is an object storage service used for storing large volumes of data, including reports.
* **Why it’s used**: Once the report is generated, it's stored in S3 for persistent storage. This allows easy access, retrieval, and sharing of reports by the client. S3 is highly durable, scalable, and cost-effective for storing reports and other large datasets.

**Possible Interview Questions & Answers:**

**1. How did you generate reports from AWS Athena, and what challenges did you face?**

* **Answer**: I wrote SQL queries to pull relevant data from Athena and then mapped the results to a custom report object. This required understanding of the data schema and the report requirements. The main challenge was ensuring that the queries were optimized for performance and that the data was structured properly for reporting purposes.

**2. Why is AWS Lambda chosen for report generation in your architecture?**

* **Answer**: AWS Lambda was chosen for its serverless nature, which allows automatic scaling based on demand. This eliminates the need to manage infrastructure, reducing operational overhead. In our case, Lambda processes reports efficiently on a per-request basis, ensuring we only pay for the compute time used.

**3. How does AWS Step Functions fit into your reporting system?**

* **Answer**: AWS Step Functions orchestrates the workflow of generating reports by coordinating different tasks such as calling AWS Lambda, checking the status of report requests in a database, and storing the generated report in S3. It ensures that each step is completed successfully before moving to the next, which is crucial for handling complex workflows.

**4. What is the role of EventBridge in your architecture?**

* **Answer**: EventBridge serves as an event bus, allowing asynchronous communication between various components of the system. When a report request is made, it triggers an event, which then triggers Lambda functions or Step Functions to process the report generation. EventBridge decouples components and ensures scalability by handling large volumes of events.

**5. Why did the system use S3 for storing the generated reports?**

* **Answer**: S3 was used for storing the generated reports because it's a highly durable, scalable, and cost-effective solution for object storage. Once the reports are generated, they can be easily stored and retrieved by the client. S3’s integration with other AWS services makes it an ideal choice for this architecture.

**6. How did you ensure the reliability and performance of the reporting system?**

* **Answer**: I made sure to optimize the SQL queries for performance and wrote unit tests with Mockito to ensure the logic around data retrieval and report generation worked correctly. While I wasn’t directly involved in the Lambda and Step Functions, I believe these services, combined with S3, provide a reliable and scalable architecture for report generation.

**7. Can you explain how you would troubleshoot if a report is not generated or stored correctly?**

* **Answer**: In case a report fails to generate or store correctly, I would first check the logs in AWS Lambda to see if there were any issues during execution. Then, I would verify the data retrieved from Athena and ensure it was mapped correctly to the custom report object. I would also review the S3 bucket for missing files and check Step Functions for any failed tasks.

**8. How would you scale the report generation system to handle more traffic?**

* **Answer**: To handle more traffic, AWS Lambda will automatically scale depending on the number of requests. Additionally, Step Functions can be used to manage retries and error handling, ensuring that the system remains resilient. If there is an increasing demand for report generation, we could consider optimizing the Athena queries further, potentially using partitions or caching mechanisms to improve performance.

**8. Client Access Security:**

* **Authentication and Authorization**:
  + **Amazon Cognito** or **OAuth**: Client access to the reports (e.g., via the CM UI) can be secured using Amazon Cognito or another authentication mechanism. This ensures that only authorized users can request and view reports.
  + **Role-Based Access Control (RBAC)**: Different users may have different roles (e.g., admin, staff, etc.), and access to reports can be controlled based on these roles.
* **API Gateway with Authorization**: If your application exposes APIs for report access, using **API Gateway** with **Cognito or IAM roles** for authentication ensures that only authorized users can call the APIs to access reports.

**Summary of Scalability Features:**

* **Lambda**: Automatically scales to handle any volume of requests, offering serverless, on-demand compute.
* **EventBridge**: Scalable event-driven architecture that decouples services and triggers processing based on demand.
* **Step Functions**: Scales workflows by executing tasks in parallel and handling retries or failovers.
* **Athena**: Serverless querying service that scales with the data size and query load.
* **S3**: Virtually unlimited storage that scales to accommodate increasing report volume.
* **CloudWatch**: Provides monitoring and alerting, enabling proactive scaling and optimization.

### **Which One is Responsible for the 5-Minute Interval?**

The **EventBridge (or CloudWatch Events)** rule is responsible for triggering the Lambda function every 5 minutes. **Step Functions** could be involved later in the workflow if there’s an orchestration of multiple tasks, but for the 5-minute interval scheduling, **EventBridge** is the correct choice.

### **How will It handle multiple requests?**

**Example Flow**:

* Lambda fetches all pending requests.
* For each request, a new Lambda invocation is launched, either by directly invoking additional Lambda functions or through services like **AWS Step Functions**, **SQS** (if queueing is involved), or **SNS** (for notification-triggered invocations).
* Each Lambda invocation handles the query, processing, and report generation for its respective request.
* Once the reports are generated, their statuses are updated, and they are stored in S3.

Future work 🡪

It sounds like your team is building an event-driven system to generate reports based on SLA calculations. Here's how the flow might look like, step by step, for generating these reports:

**Application Flow for SLA Calculation and Report Generation**

**1. Consuming Events from Kafka running in ECS**

* **Kafka Connect**: The system will consume events from Kafka that contain the required data for the SLA report. These events will include:
  + **First Timestamps**: Time when the request was first initiated or started.
  + **Request ID**: Unique identifier for each request.
  + **Email Address / SMS Number**: Contact information associated with the request.
  + **SLA Criteria**: Information about the service level agreement, such as the expected time limits for the request to be completed.

**2. Storing Event Data**

* The consumed Kafka events will be processed, and relevant fields (request ID, first timestamp, email/SMS, SLA criteria) will be stored temporarily or in an event store, depending on your design.

### **2. Amazon SQS (Simple Queue Service)**

* **Cost Structure**:
  + **Request Costs**: SQS charges based on the number of requests made to the service. The first **1 million requests** are free each month, and after that, it’s **$0.40 per million requests**.
  + **Data Transfer**: Data transfer into and out of SQS is free within the same region. If you transfer data across regions, it could incur additional costs.
  + **Message Retention**: SQS can retain messages for up to 14 days. You only pay for the messages in the queue, which can help keep costs low if messages aren’t stored for long.
  + **Cost Considerations**:
  + **Very Cost-Effective for Low Volume**: If the volume of messages is low or if you have short-lived events, SQS is an inexpensive way to buffer data.
  + **Scalable and Pay-As-You-Go**: Costs scale with the number of requests, so it can be extremely cost-effective for bursty workloads.

**3. Retrieving Last Timestamps from the Database**

* At the end of the day, the system will retrieve the **last timestamp** for each request from the database. This timestamp will be compared with the **first timestamp** (from the Kafka event) to calculate SLA.

**4. SLA Calculation**

* **SLA Logic**: Based on the first timestamp (from Kafka) and the last timestamp (from the database), calculate the SLA compliance (e.g., whether the request was completed within the agreed SLA time).
  + **SLA Metrics**: Depending on the SLA criteria, the SLA could be calculated based on response time, resolution time, or any other relevant metrics.
  + **Aggregation**: You may need to aggregate SLA data per user, per request, or across various time periods (e.g., daily, weekly) to get insights about SLA adherence.

**5. Report Generation**

* **Report Data**: Using the calculated SLA data, generate the report. This could include:
  + **Total Requests Processed**: Number of requests handled during the day.
  + **Requests Within SLA**: Number of requests completed within SLA.
  + **Requests Outside SLA**: Number of requests exceeding the SLA time.
  + **Customer Contact Information**: Email/SMS data for users who had SLA violations or for other purposes like notification.
* **Report Format**: The report could be generated in a structured format, such as **JSON**, **CSV**, or **PDF**, depending on your reporting requirements.

**6. Storing the Report in S3**

* Once the report is generated, it will be stored in **Amazon S3** for persistent storage. Each report can be stored with a unique name based on the date, request ID, or other identifiers, to ensure that reports are easily retrievable.
  + **S3 Security**: Ensure the report is securely stored with appropriate **S3 encryption** and **access policies** to control who can read and write the reports.
  + **Versioning**: If required, enable **S3 versioning** to track different versions of the reports over time.

**7. Notifications (Optional)**

* Once the report is stored in S3, you can use **SNS** or **EventBridge** to send notifications to users or systems indicating that the SLA report is available.
  + **Email or SMS Notifications**: You could send an email or SMS to notify relevant stakeholders or customers about their SLA performance.

**8. Monitoring and Logging**

* **CloudWatch Logs**: Log the event consumption, SLA calculations, and report generation steps to **Amazon CloudWatch Logs** for monitoring and debugging.
* **CloudTrail**: Track API calls related to S3 storage, Lambda invocations, and other AWS services used in the workflow.

**Application Flow Summary:**

1. **Kafka Event Consumption**: Kafka events containing request details (first timestamp, request ID, SLA criteria, etc.) are consumed.
2. **Store Event Data**: The event data is stored temporarily or in an event store for further processing.
3. **Retrieve Last Timestamps**: The system queries the database at the end of the day to retrieve the last timestamps.
4. **SLA Calculation**: SLA is calculated by comparing the first timestamp (from Kafka) and last timestamp (from DB) based on SLA criteria.
5. **Report Generation**: A report is generated containing SLA metrics and user information.
6. **Store Report in S3**: The generated report is stored securely in **Amazon S3**.
7. **Optional Notification**: After storing the report, notifications are sent via **SNS** or **EventBridge** to alert users about report availability.
8. **Monitoring and Logging**: Use **CloudWatch Logs** and **CloudTrail** to track the execution flow and debug if needed.

**How to Explain This Flow in an Interview:**

You can explain how your team’s system consumes events from **Kafka**, processes the event data (timestamps, request IDs, SLA criteria), and compares it with data stored in the **database** to calculate **SLA compliance**. The **report generation** is done at the end of the day, and the report is stored securely in **Amazon S3**. If necessary, notifications are sent to stakeholders to alert them about the reports generated. All of this is monitored via **CloudWatch** for performance tracking.

Resume points 🡪

* Developed code to run dynamic SQL queries to retrieve counts and data from AWS Athena for report generation by leveraging AWS services like **Lambda**, **EventBridge**, and **Step Functions** for seamless orchestration.
* Contributed to the design of the end-to-end data flow, ensuring smooth integration between Athena, data processing, and reporting and worked with the team to ensure data accuracy.
* Collaborated with the team to plan a new design for SLA report generation system, involving Kafka event consumption and SLA calculations based on first and last timestamps.
* worked with the architect for plan a scalable, event-driven system for SLA reports, Led POC developments on Kafka event consumption, AWS s3 storage integration.
* Assisted in designing the logic for efficient data mapping and transforming Athena results into actionable insights.

## CCIL 🡪

CCIL (Customer Communication Integration Layer) is a set of integrated services providing digital communication capabilities.

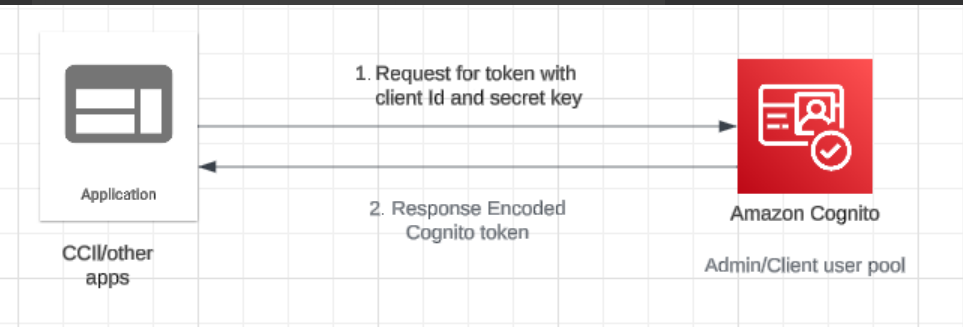
This is client committed work that is funded. There are contractual commitments that need to be met from a service delivery perspective.

Order 🡪 list of documents 🡪 actions (compose, store, retrieve, merge, e-deliver)

### AWS Cognito (Authentication)🡪

**Summary:** CCIL aws account has 2 types of cognito user pools

* Admin cognito pool - pool where admins can access to perform admin/secure operations with ADMIN\_CLIENT created at the time of pool creation. Used to on-aboard app clients and provide required/limited access to application services
* App client pool - user pool where all applications/clients to the services resides with respective scopes associated to the API gateway services.

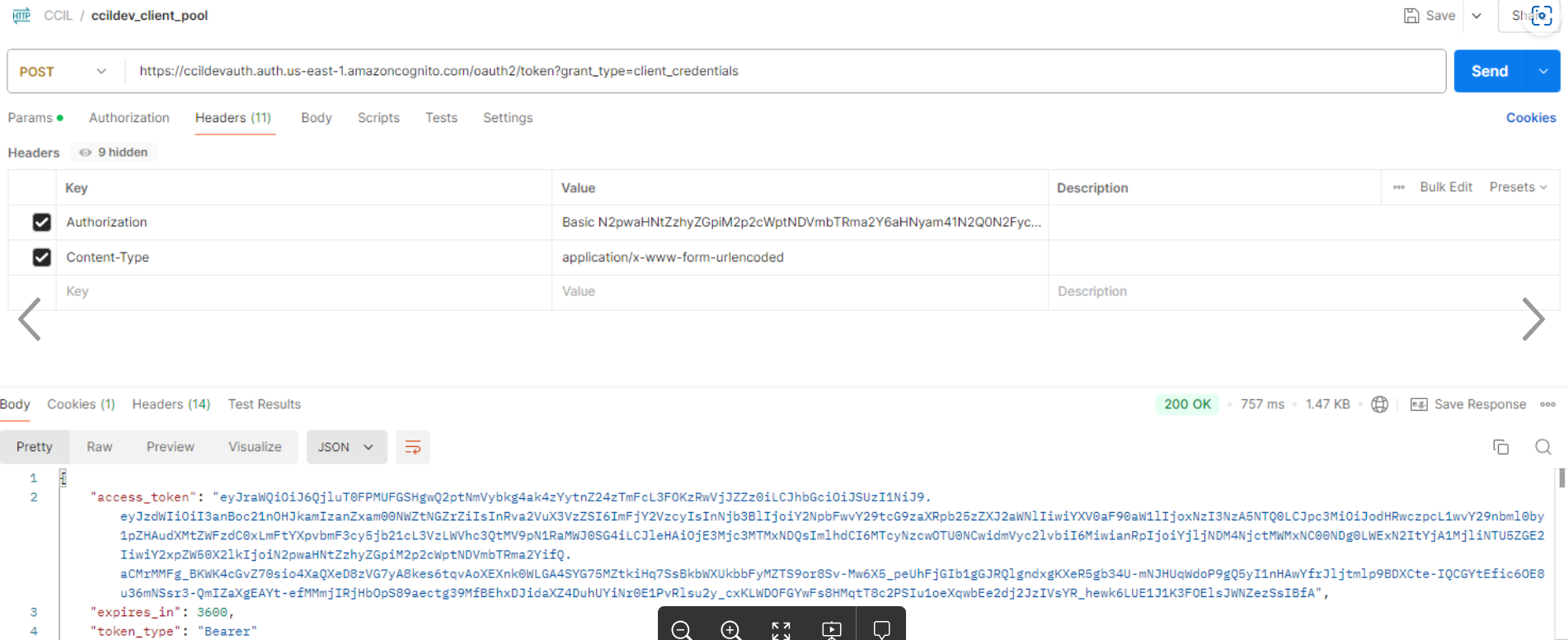


**Cognito Authentication:**

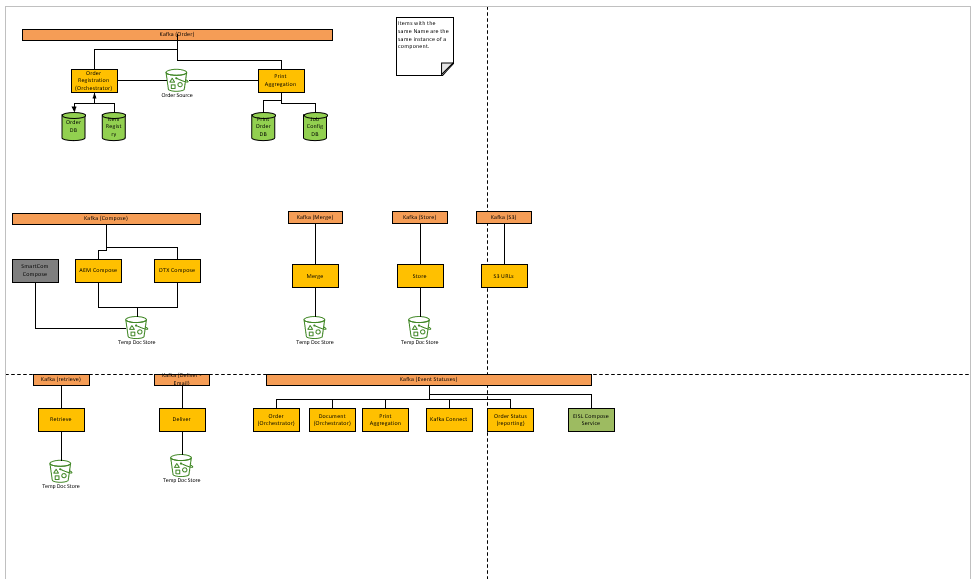
| **Userpool** | **Authentication Urls** | **Headers** |
| --- | --- | --- |
| Admin pool **(br-ics{$sdlc}-ccil-admin-pool)** | <https://ccildevadmin.auth.us-east-1.amazoncognito.com/oauth2/token?grant_type=client_credentials> | Authorization: Basic  (Base64 encoded clientid:secretkey)  Content-type: application/x-www-form-urlencoded  grant\_type: client\_credentials  HTTP Method: POST |
| App Client pool **(br-ics{$sdlc}-ccil-appclient-pool)** | <https://ccildevauth.auth.us-east-1.amazoncognito.com/oauth2/token?grant_type=client_credentials> | Authorization: Basic  (Base64 encoded clientid:secretkey)  Content-type: application/x-www-form-urlencoded  grant\_type: client\_credentials  HTTP Method: POST |

**Steps to get the Token Generated:**

* Get the[ADMIN\_CLIENT](https://us-east-1.console.aws.amazon.com/cognito/v2/idp/user-pools/us-east-1_r227VvLam/app-integration/clients/1eq60v545mhq5l3ssi3llloedb?region=us-east-1) / APP\_CLIENT clientId and secretId from the user pools
* Base64 Encode the client Id and secret key and use the encoded value in the Authorization Header like "**Basic  {Base64 encoded clientid:secretkey}**".
* Use that urls mentioned above to generate the cognito tokens



### Topic view 🡪



**ote-topic-name:** br.ccil.cmp.oterequest.1  
**storage-topic-name:** br.communication.document.storage.ppg.0  
**document-topic-name:** br.communication.document.document.ppg.0  
**composition-aem-topic-name:** br.communication.composition.aem.0  
**merge-topic-name:** br.communication.document.merge.ppg.0  
**delivery-topic:** br.communication.messages.commands.0  
**document-serviceaction-status-topic-name:** br.communication.document.serviceactionstatus.ppg.0  
**order-status-topic-name:** br.communication.order.status.commands.0  
**notifications-topic:** br.communication.documents.notifications.0  
**retrieve-input-topic-name:** br.communication.document.retrieve.0

### Json structures🡪

[Principal Shared Assets - Principal Financial Group - Confluence\_AWS\_Prod](https://confluence.broadridge.net/spaces/ICSPFG/pages/686032674/Principal+Shared+Assets)

#### Order Topic 🡪

{  
 "orderId": "0044AM",  
 "communicationCommon": {  
 "communicationEntity": "PFG",  
 "communicationParty": [  
 {  
 "partyRole": "Requestor",  
 "partyName": "Automation",  
 "partyIdentifierType": "requestorId",  
 "partyIdentifierValue": "XAASUNS\_Auto"  
 }  
 ],  
 "communicationIdentifierType": "requestId",  
 "communicationID": [  
 {  
 "communicationIdentifierType": "distributionId"  
 },  
 {  
 "communicationIdentifierType": "jobName"  
 }  
 ],  
 "communicationNarrativeType": "copyCount",  
 "communicationClassificationType": "emailDeliveryApplication",  
 "communicationFlags": {  
 "item": {  
 "communicationFlagType": "isForeignAddress",  
 "communicationFlagValue": false  
 }  
 },  
 "nonStandardItems": {  
 "businessUnitType": "CSS",  
 "departmentId": "H817000",  
 "notificationProperties": {  
 "topicSubscriptionKey": "btesttsk",  
 "userDefinedProperties": [  
 {  
 "userDefinedProperty": {  
 "key": "test",  
 "value": "more detail"  
 }  
 }  
 ]  
 }  
 }  
 },  
 "communicationDocuments": [  
 {  
 "documentId": "DOC1",  
 "communicationDocumentSequenceNumber": 1,  
 "communicationActions": [  
 {  
 **"communicationActionType": "composition",**  
 "communicationActionSequenceNumber": 1,  
 "compositionData": "\"<?xml version=\\\"1.0\\\" encoding=\\\"UTF-8\\\"?><Request xmlns:xsi=\\\"http://www.w3.org/2001/XMLSchema-instance\\\" xsi:noNamespaceSchemaLocation=\\\"LFWBatchRequestXMLSchema.xsd\\\"><Letters><CompositionVariables> <IMAGE>Logo</IMAGE> <letter>PIN</letter> <Date>October 04, 2021</Date> <AfltId>1234567890123456789</AfltId> <!-- Affiliate id--> <highProfileClient>N</highProfileClient> <!-- High profile client indicator--> <mailFlow>P-001-N10</mailFlow> <!-- mailflow for return mail--> <docId>GP59269-0</docId> <updateSource>on our website</updateSource> <!-- updated via channel--> <firstName>John</firstName> <lastName>Doe</lastName> <raddr1>Principal</raddr1> <raddr2>711 High Street</raddr2> <raddr3>Des Moines IA 50392</raddr3> <raddr4></raddr4> <raddr5></raddr5> <raddr6></raddr6> <addr1>John Doe</addr1> <addr2>47474747 PART CT</addr2> <addr3>CREEDMOOR, NC 74747-777</addr3> <addr4></addr4> <addr5></addr5> <addr6></addr6> <chrgDept>S701000</chrgDept> <phone>800-986-3343</phone></CompositionVariables></Letters></Request>",  
 "communicationActionRegistry": [  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKey",  
 "communicationCustomValue": "3067"  
 },  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKeyVersion",  
 "communicationCustomValue": 1  
 }  
 ]  
 }  
 ],  
 "nonStandardItems": {  
 "documentDetails": {  
 "businessUnitDocumentInstanceId": "1234",  
 "documentSequence": "1"  
 }  
 }  
 },  
 {  
 "documentId": "DOC2",  
 "communicationDocumentSequenceNumber": 2,  
 "communicationActions": [  
 {  
 "communicationActionType": "composition",  
 "communicationActionSequenceNumber": 1,  
 "compositionData": "\"<?xml version=\\\"1.0\\\" encoding=\\\"UTF-8\\\"?><Request xmlns:xsi=\\\"http://www.w3.org/2001/XMLSchema-instance\\\" xsi:noNamespaceSchemaLocation=\\\"LFWBatchRequestXMLSchema.xsd\\\"><Letters><CompositionVariables> <IMAGE>Logo</IMAGE> <letter>PIN</letter> <Date>October 04, 2021</Date> <AfltId>1234567890123456789</AfltId> <!-- Affiliate id--> <highProfileClient>N</highProfileClient> <!-- High profile client indicator--> <mailFlow>P-001-N10</mailFlow> <!-- mailflow for return mail--> <docId>GP59269-0</docId> <updateSource>on our website</updateSource> <!-- updated via channel--> <firstName>John</firstName> <lastName>Doe</lastName> <raddr1>Principal</raddr1> <raddr2>711 High Street</raddr2> <raddr3>Des Moines IA 50392</raddr3> <raddr4></raddr4> <raddr5></raddr5> <raddr6></raddr6> <addr1>John Doe</addr1> <addr2>47474747 PART CT</addr2> <addr3>CREEDMOOR, NC 74747-777</addr3> <addr4></addr4> <addr5></addr5> <addr6></addr6> <chrgDept>S701000</chrgDept> <phone>800-986-3343</phone></CompositionVariables></Letters></Request>",  
 "communicationActionRegistry": [  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKey",  
 "communicationCustomValue": "3067"  
 },  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKeyVersion",  
 "communicationCustomValue": 1  
 }  
 ]  
 }  
 ],  
 "nonStandardItems": {  
 "documentDetails": {  
 "businessUnitDocumentInstanceId": "79888",  
 "documentSequence": "2"  
 }  
 }  
 },  
 {  
 "documentId": "66666",  
 "communicationDocumentSequenceNumber": 3,  
 "nonStandardItems": {  
 "documentDetails": {  
 "businessUnitDocumentInstanceId": "1234",  
 "documentSequence": "3",  
 "staticDocument": {  
 "businessIdentifier": "DD983-06"  
 }  
 }  
 }  
 },  
 {  
 "documentId": "113323",  
 "communicationDocumentIdentifierType": "combinedDocument",  
 "communicationDocumentIdentifierValue": true,  
 "communicationDocumentSequenceNumber": 4,  
 "communicationActions": [  
 {  
 "communicationActionType": "storageRetention",  
 "communicationActionSequenceNumber": 2,  
 "communicationActionRegistry": [  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKey",  
 "communicationCustomValue": "DOC4"  
 },  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKeyVersion",  
 "communicationCustomValue": 1  
 }  
 ],  
 "communicationIndices": [  
 {  
 "communicationCustomKey": "DOC\_INSTC\_ID",  
 "communicationCustomValue": "SOMEDOC"  
 },  
 {  
 "communicationCustomKey": "RM\_DOC\_TYP\_TXT",  
 "communicationCustomValue": "Temporary Document"  
 }  
 ],  
 "nonStandardItems": {  
 "linearize": "false",  
 "compress": "false"  
 }  
 }  
 ],  
 "nonStandardItems": {  
 "documentDetail": {  
 "businessUnitDocumentInstanceId": "1233",  
 "documentSequence": "1"  
 }  
 }  
 }  
 ]  
 }

#### XML 🡪 for pin letter

<?xml version=\\\"1.0\\\" encoding=\\\"UTF-8\\\"?>  
<Request xmlns:xsi=\\\"http://www.w3.org/2001/XMLSchema-instance\\\" xsi:noNamespaceSchemaLocation=\\\"LFWBatchRequestXMLSchema.xsd\\\">  
 <Letters>  
 <CompositionVariables>  
 <IMAGE>Logo</IMAGE>  
 <letter>PIN</letter>  
 <Date>October 04, 2021</Date>  
 <AfltId>1234567890123456789</AfltId>  
 <!-- Affiliate id-->  
 <highProfileClient>N</highProfileClient>  
 <!-- High profile client indicator-->  
 <mailFlow>P-001-N10</mailFlow>  
 <!-- mailflow for return mail-->  
 <docId>GP59269-0</docId>  
<updateSource>on our website</updateSource>  
 <!-- updated via channel-->  
 <firstName>John</firstName>  
 <lastName>Doe</lastName>  
 <raddr1>Principal</raddr1>  
 <raddr2>711 High Street</raddr2>  
<raddr3>Des Moines IA 50392</raddr3>  
 <raddr4></raddr4>  
 <raddr5></raddr5>  
 <raddr6></raddr6>  
<addr1>John Doe</addr1>  
 <addr2>47474747 PART CT</addr2>  
 <addr3>CREEDMOOR, NC 74747-777</addr3>  
 <addr4></addr4>  
 <addr5></addr5>  
 <addr6></addr6>  
 <chrgDept>S701000</chrgDept>  
 <phone>800-986-3343</phone>  
 </CompositionVariables>  
 </Letters>  
</Request>

A screenshot of a computer

AI-generated content may be incorrect.

#### Order Status Topic 🡪

br.communication.order.status.commands.0 (status – NOT-STARTED, FAILED, COMPLETED)

Key -- null1159PM

Value --

{  
 "orderId": "1159PM",  
 "documents": [  
 {  
 "documentId": "DOC1",  
 "status": "NOT-STARTED",  
 "combineDocument": false  
 },  
 {  
 "documentId": "DOC2",  
 "status": "NOT-STARTED",  
 "combineDocument": false  
 },  
 {  
 "documentId": "113323",  
 "status": "NOT-STARTED",  
 "combineDocument": true  
 }  
 ],  
 "status": null  
}

#### Document Topic 🡪

br.communication.document.document.ppg.0

{  
 "createdDate": null,  
 "expirytime": null,  
 "status": null,  
 "orderId": "5325810",  
 "communicationDocumentSequenceNumber": 1,  
 "externalId": "5325810",  
 "documentId": "5325810doc",  
 "communicationActions": [  
 {  
 "status": null,  
 "documentRegistry": null,  
 "communicationActionType": "RETRIEVE",  
 "communicationActionSequenceNumber": 1,  
 "documentId": "5325810doc",  
 "communicationParties": null,  
 "compositionData": null,  
 "communicationActionRegistry": [  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKey",  
 "communicationCustomValue": "3703"  
 },  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKeyVersion",  
 "communicationCustomValue": "1"  
 }  
 ],  
 "communicationIndices": null,  
 "communicationDetail": null,  
 "s3Locator": "PFG/5325810/5325810/5325810doc",  
 "resultS3Locator": null,  
 "attachementS3Locators": null,  
 "s3LogFileKey": null,  
 "communicationDocumentRegistryIdentifier": null,  
 "communicationDocumentRegistryIdentifierVersion": null,  
 "attachments": null,  
 "communicationActionId": null,  
 "mergeItems": null,  
 "nonStandardItems": null,  
 "requestedDocumentID": null,  
 "composedEmailS3Locator": null,  
 "compositionEngine": null,  
 "dataFormatType": null  
 },  
 {  
 "status": null,  
 "documentRegistry": null,  
 "communicationActionType": "composition",  
 "communicationActionSequenceNumber": 2,  
 "documentId": "5325810doc",  
 "communicationParties": null,  
 "compositionData": "{\"CSSDR\_Test\": { \"OS\_Env\": \"MVS\",\"Var1\": \"Regression\",\"Var2\": \"Testing\",\"Var3\": \"11 December 2014\",\"Var4\": \"12:35\",\"Var5\": \"Variable 5\",\"Var6\": \"Variable 6\",\"Var7\": \"Variable 7\",\"Var8\": \"Variable 8\",\"Var9\": \"Variable 9\",\"Var10\": \"Variable 10\",\"RTFFilePath\": \"\"}}",  
 "communicationActionRegistry": [  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKey",  
 "communicationCustomValue": "3067"  
 },  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKeyVersion",  
 "communicationCustomValue": "1"  
 }  
 ],  
 "communicationIndices": null,  
 "communicationDetail": null,  
 "s3Locator": "PFG/5325810/5325810/5325810doc",  
 "resultS3Locator": null,  
 "attachementS3Locators": null,  
 "s3LogFileKey": null,  
 "communicationDocumentRegistryIdentifier": null,  
 "communicationDocumentRegistryIdentifierVersion": null,  
 "attachments": null,  
 "communicationActionId": null,  
 "mergeItems": null,  
 "nonStandardItems": null,  
 "requestedDocumentID": null,  
 "composedEmailS3Locator": null,  
 "compositionEngine": null,  
 "dataFormatType": null  
 },  
 {  
 "status": null,  
 "documentRegistry": null,  
 "communicationActionType": "storageRetention",  
 "communicationActionSequenceNumber": 3,  
 "documentId": "5325810doc",  
 "communicationParties": null,  
 "compositionData": null,  
 "communicationActionRegistry": [  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKey",  
 "communicationCustomValue": "3703"  
 },  
 {  
 "communicationCustomKey": "RegistryItemDetailItemKeyVersion",  
 "communicationCustomValue": "1"  
 }  
 ],  
 "communicationIndices": [  
 {  
 "communicationCustomKey": "DOC\_INSTC\_ID",  
 "communicationCustomValue": "2002"  
 },  
 {  
 "communicationCustomKey": "RM\_DOC\_TYP\_TXT",  
 "communicationCustomValue": "Temporary Document"  
 }  
 ],  
 "communicationDetail": null,  
 "s3Locator": "PFG/5325810/5325810/5325810doc",  
 "resultS3Locator": null,  
 "attachementS3Locators": null,  
 "s3LogFileKey": null,  
 "communicationDocumentRegistryIdentifier": null,  
 "communicationDocumentRegistryIdentifierVersion": null,  
 "attachments": null,  
 "communicationActionId": null,  
 "mergeItems": null,  
 "nonStandardItems": {  
 "staticFormInput": null,  
 "documentDetails": null,  
 "linearize": false,  
 "compress": false  
 },  
 "requestedDocumentID": null,  
 "composedEmailS3Locator": null,  
 "compositionEngine": null,  
 "dataFormatType": null  
 }  
 ],  
 "clientId": "PFG",  
 "clientEchoback": {  
 "businessunit": "CSS",  
 "requestid": "2002",  
 "notificationProperties": {  
 "topicSubscriptionKey": "kbtest",  
 "userDefinedProperties": null  
 }  
 },  
 "nonStandardItems": {  
 "staticFormInput": null,  
 "documentDetails": {  
 "businessUnitDocumentInstanceId": "2002",  
 "documentSequence": "1",  
 "staticDocument": null  
 },  
 "linearize": null,  
 "compress": null  
 }  
 }

### Jgdls

### Hbfjkksd

 **Order and Document Services (Kafka, DynamoDB, KTables)**

* This project focuses on event-driven architecture using Kafka and stateful processing with KTables. You’ve also worked with DynamoDB for persistence and managed the orchestration of service actions (e.g., Compose and Store). This type of project demonstrates your expertise in event-driven systems, microservices architecture, and cloud-native services (such as Kafka, DynamoDB, etc.), making it a great example to showcase your experience, especially if you're applying for roles related to distributed systems, cloud services, or event-driven architecture.

 **Report Generation System (EventBridge, Lambda, Athena, S3)**

* This project is focused on serverless computing, event-driven architecture with AWS services, and report generation. It demonstrates your ability to work with AWS serverless technologies (EventBridge, Lambda, Athena, S3) and your skills in data processing and report generation. If you're applying for roles involving AWS cloud services, serverless architecture, or data processing, this would be a great project to highlight.