Project: SwiftLine Logistics AI Chatbot

Document Type: Architectural Design & Technical Rationale

1. Executive Summary

The SwiftLine AI Chatbot is a serverless, event-driven solution designed to automate Level 1 order tracking enquiries. By leveraging Amazon Lex V2 for Natural Language Understanding (NLU) and AWS Lambda for backend fulfillment, the system allows customers to retrieve real-time order status 24/7 without human intervention. The infrastructure is fully automated using a Hybrid Infrastructure-as-Code (IaC) strategy, utilizing Terraform for core resources and CloudFormation for bot definitions, ensuring a secure, reproducible, and scalable environment.

2. Architectural Decision & Rationale

A. Compute: AWS Lambda (Python 3.12)

- **Decision**: Serverless backend logic rather than EC2/Containers.
- **Rationale**: Order tracking traffic is sporadic and "bursty". Lambda's event-driven model eliminates idle server costs (paying only for milliseconds of execution) and removes the operational overhead of OS patching and server management.

B. Data: Amazon DynamoDB (On-demand)

- **Decision**: NoSQL storage with trackingId as the partition key.
- **Rationale**: The data structure is hierarchical (Customer -> Delivery -> Items) and fits a JSON-document model perfectly. The "On-Demand" capacity mode allows the database to instantly scale to handle peak logistics seasons without manual provisioning while providing single-digit millisecond latency for lookups.

C. Infrastructure as Code: Hybrid Strategy

- **Decision**: Terraform manages core infrastructure (IAM, Lambda, DB), while CloudFormation manages Lex resources.
- **Rationale**: The native Terraform provider for Lex V2 is verbose and structurally complex. By wrapping a CloudFormation template inside a Terraform resource, we leverage Terraform's superior state management for the "plumbing" and CloudFormation's native stability for the "brain" (Lex definition).

D. Observability: Dual-Layer Logging

- **Decision**: Centralized CloudWatch logging for both application logic and conversation text.
- **Rationale**: We enabled *ConversationLogSettings* on the Bot Alias to stream full-text inputs. This allows engineers to audit "missed intents" and tune NLU accuracy, while Lambda logs capture runtime errors.

3. Security & IAM Strategy (Least Privilege)

This solution adheres strictly to the AWS Well-Architected Framework:

- **Identity Management**: Granular IAM roles were created instead of broad "full access" policies.
- **Lambda Scope**: The execution role is restricted to *dynamodb:GetItem* and *dynamodb:Query* only on the specific *swiftline-orders* table ARN. It cannot scan the database or access other resources.
- **Lex Scope:** The service role is restricted to writing to its specific log group and invoking only the designated fulfillment Lambda alias.

4. Trade-off Analysis

A. API Gateway vs. Console Testing

- **Trade-off**: Building a public Web UI provides a better demo but introduces security risks (public access) and complexity (CORS, Cognito).
- **Resolution**: I prioritized backend security. Testing is conducted via the IAM-secured AWS Console. This avoids creating an unsecured public endpoint while effectively demonstrating the backend architecture.

B. DynamoDB vs. RDS

- **Trade-off**: RDS offers complex ad hoc SQL queries; DynamoDB offers speed and scale.
- **The Rationale**: Since the access pattern is strictly "Get Order by ID", DynamoDB was the more performant and cost-effective choice. We traded query flexibility for speed and lower overhead.

5. Addressing the Business Problem

- **Eliminating Support Bottlenecks:** currently, support staff are overwhelmed by high volumes of "Where is my order?" emails. This solution automates 100% of these Level 1 enquiries, allowing the support team to focus on complex resolutions rather than data lookups.

- **Reducing Customer Wait Times:** Instead of waiting hours or days for an email reply, customers receive instant, real-time status updates 24/7.
- **Preventing User Frustration:** By implementing input sanitization (auto-uppercasing tracking IDs) and fuzzy matching logic, the bot resolves enquiries that would otherwise fail due to minor typos, preventing customers from defaulting back to email support.

6. Potential Improvements for Production

To take this MVP (Minimum Viable Product) to a global production rollout, the following enhancements are recommended:

1. **Frontend Integration (Amazon Lex Web UI)**: Deploy the Amazon Lex Web UI (a pre-built CloudFormation solution) to host a React-based chat widget on S3/CloudFront. This would provide the customer-facing interface required for public access.
2. **User Authentication (Amazon Cognito):** Integrate Amazon Cognito to authenticate users before the chat begins. This allows the bot to access the user's profile and answer "Where is my order?" without requiring the user to manually type a tracking ID.
3. **Omni-channel Deployment:** Configure Amazon Pinpoint or Twilio integrations to allow the bot to respond via SMS or WhatsApp, meeting customers on the platforms they already use.
4. **Customer Support Redirect:** Integrate with Amazon Connect. If the bot detects negative sentiment or if the user clicks "No" when asked if they are satisfied, the session context should be transferred to a live human agent.