Stage 1 — Prerequisites & Environment Setup

Goal

Set up your AWS environment, permissions, and foundational configurations so everything else works securely and smoothly.

What We Do

- Create and configure your **AWS account**.
- Enable **Amazon Bedrock** and request access to **Claude** (for conversation) and **Titan Embeddings** (for retrieval/knowledge-base).
- Get a **data API key** (from Alpha Vantage, Finnhub, or Nasdaq Data Link).
- Store credentials safely in **AWS Secrets Manager**.
- Prepare IAM roles so Bedrock Agents and Lambda functions can communicate securely.

Why We Use These Services

Service	Purpose
AWS IAM	Controls access to each AWS service and ensures least-privilege permissions.
AWS Secrets Manager	Stores sensitive API keys securely, instead of hardcoding them.
Amazon Bedrock	Provides foundational AI models like Claude for conversational understanding.
External API	Supplies real-time and historical market data.

Languages / Tools

- **AWS Console** or **CLI** for setup
- **Python (boto3)** or any for testing connections and fetching secrets
- **Terraform / CDK (optional)** for infrastructure as code

Step-by-Step Guide

1. **Create AWS Account** and set up billing alerts.
2. **Create IAM Role** named, for example, `StockChatbotRole` with Bedrock and Lambda access.
3. **Request Model Access** in Amazon Bedrock (Claude + Titan).
4. **Generate API Key** from Alpha Vantage or Finnhub.
5. **Store API Key** in Secrets Manager under a name like `/api/key`.
6. **Test** with a simple Python script to ensure you can retrieve secrets.
7. **Set Up S3 Bucket** (optional) for storing logs and stock data backups.

Stage 2 — Stock Data Handling & Analysis Logic

Goal

Create the backend logic to fetch stock data, analyze it technically, and predict future trends.

What We Do

Build **three Lambda functions**:

1. **Data Fetcher** — retrieves stock prices from an external API.
2. **Technical Analyzer** — calculates indicators like RSI, SMA, EMA.
3. **Prediction Model** — optionally invokes a SageMaker model for forecasting.

All data can be stored in **Amazon S3** for analysis and record-keeping.

Why We Use These Services

Service	Purpose
AWS Lambda	Serverless compute for quick, scalable backend functions.
Amazon S3	Durable, inexpensive data lake for historical stock data.

Amazon SageMaker	Optional ML service to host and deploy stock prediction models.
AWS CloudWatch	For logging and debugging each Lambda’s activity.

Languages / Tools

- **Language:** Python 3.11
- **Libraries:** `requests`, `boto3`, `pandas`, `numpy`, `pandas_ta` (for indicators)
- **Environment:** AWS Lambda Console or local VS Code + AWS SAM CLI

Step-by-Step Guide

Lambda 1 — Data Fetcher

Goal: Fetch stock data in real-time.
Steps:

1. Create Lambda function `DataFetcherLambda`.
2. Attach IAM permissions for Secrets Manager + S3 write access.
3. In Python, retrieve API key from Secrets Manager.
4. Call API (e.g., Alpha Vantage `TIME_SERIES_DAILY_ADJUSTED`).
5. Parse the JSON response.
6. Store data to S3 or return it directly.

Example Output:

```
{"symbol":"TSLA","date":"2025-10-10","close":253.21}
```

Lambda 2 — Technical Analyzer

Goal: Compute indicators like RSI, SMA, EMA.
Steps:

1. Create `TechnicalAnalyzerLambda`.
2. Input: stock symbol or data from S3.
3. Use `pandas` or `pandas_ta` to compute requested indicator.
4. Return a value and interpretation (e.g., “RSI 72 → overbought”).

Example Output:

```
{"symbol":"AAPL","indicator":"RSI","value":68.45,"signal":"Approaching
overbought zone"}
```

Lambda 3 — Prediction Model

Goal: Predict next-day or next-week trends.
Steps:

1. Train or deploy a model in **Amazon SageMaker**.
2. Create `PredictorLambda` that calls the SageMaker endpoint.
3. Pass preprocessed features (e.g., last 30 days' prices).
4. Receive prediction (e.g., "+2% expected growth").

Example Output:

```
{"symbol":"MSFT","prediction":"Likely upward trend next week (+1.8%)"}
```

Stage 3 — Agent Orchestration with Amazon Bedrock

Goal

Enable the chatbot to understand natural language queries and automatically decide which Lambda function to invoke.

What We Do

Create an **Amazon Bedrock Agent** that acts as the chatbot's brain.
We register each Lambda as an **Action Group**, define its schema, and instruct the Agent when to use which tool.

Why We Use These Services

Service	Purpose
Amazon Bedrock Agent	Interprets user requests and orchestrates Lambda calls.
OpenAPI Schemas	Tell the agent how to use each Lambda function.
IAM Role	Grants Agent permission to invoke Lambdas securely.

Languages / Tools

- **OpenAPI (YAML or JSON)** for Action Group definitions
- **AWS Console or Bedrock SDK**
- **Python (boto3)** to test `invoke_agent()`

Step-by-Step Guide

1. Open Amazon Bedrock → "Create Agent."
2. Choose a foundation model (e.g., **Claude v3**).
3. Attach an **IAM role** that allows `lambda:InvokeFunction`.
4. Define **Action Groups**:
    - `fetchStockData` → DataFetcherLambda
    - `calculateIndicator` → TechnicalAnalyzerLambda
    - `predictStockTrend` → PredictorLambda
5. Provide **OpenAPI schema** for each Lambda (inputs, outputs).
6. Configure **Agent Instructions** (prompt):
   "You are a professional virtual stock analyst. Fetch data for real-time queries and compute indicators when requested."
7. **Test** in the console:
   Ask → "What's the 10-day RSI for Google stock?"
   The agent calls the correct Lambda and summarizes results.

Stage 4 — User Interface (Frontend + API Layer)

Goal

Let users chat interactively with your Bedrock Agent through a web app.

What We Do

We create:

1. **API Gateway Endpoint** – receives user queries and forwards them to Bedrock.
2. **Lambda Proxy** – formats and sends requests to `invoke_agent()` API.
3. **Frontend App** – chat UI for typing questions and seeing responses.
4. **Hosting** – serve frontend securely via S3 and CloudFront.

Why We Use These Services

Service	Purpose
Amazon API Gateway	Securely exposes API endpoint to frontend.
AWS Lambda (Proxy)	Handles Bedrock call logic and response formatting.
Amazon S3 + CloudFront	Hosts and delivers frontend quickly.
React / Streamlit	Provides simple interactive chat UI.

Languages / Tools

- **Backend:** Python for proxy Lambda
- **Frontend:** React (JS/TypeScript) or Streamlit (Python)
- **Hosting:** S3 + CloudFront
- **API Calls:** `fetch()` or `axios`

Step-by-Step Guide

1. Create **Lambda Proxy** to call Bedrock's `invoke_agent()` and return JSON.
2. Create **API Gateway** with POST endpoint `/chat`.
3. Connect API Gateway to Lambda Proxy and enable CORS.
4. Build a **Frontend UI**:
   - Chat input box
   - Response display area
   - Message history
5. Deploy frontend files to **S3**.
6. Configure **CloudFront** for HTTPS and global caching.
7. Test:
   - Type "What's the stock price of Tesla?"
   - See Bedrock Agent respond with live data.

Stage 5 — Monitoring, Security & Scaling

Goal

Ensure system reliability, monitor errors, secure data, and handle large workloads efficiently.

What We Do

Add observability, authentication, and scalability to production-ready standards.

Why We Use These Services

Service	Purpose
CloudWatch	Logs, metrics, and alarms for Lambda and API Gateway.
Cognito	Adds login/user authentication for API access.
DynamoDB	Stores chat history or user sessions.
Auto Scaling / Fargate	Handles high loads automatically.

Languages / Tools

- **CloudWatch Logs & Metrics**
- **DynamoDB (NoSQL)**
- **Cognito User Pools**
- **AWS CLI or Console**

Step-by-Step Guide

1. **Enable CloudWatch Logs** for all Lambdas and API Gateway.
2. Create **CloudWatch Alarms** (errors, latency, throttles).
3. Add **Cognito Authentication** to API Gateway if required.
4. Use **DynamoDB** to store chat history for context retention.
5. Enable **Auto Scaling** for Lambdas (concurrency) or ECS tasks.
6. Periodically **review IAM roles** for least privilege.
7. Implement **cost monitoring** with AWS Budgets.

Final Architecture Summary

User → Frontend (React/Streamlit)
→ API Gateway (REST endpoint)
→ Lambda Proxy
→ Bedrock Agent (Claude)
→ Action Groups (Lambdas for Data, Analysis, Prediction)
→ External APIs / S3 / SageMaker
→ Back to User Response