Building a Weather Application with GenAI, Spring Boot, and Microservices

1. Introduction

The integration of Artificial Intelligence (AI), particularly Generative AI (GenAI), is increasingly transforming the landscape of software development. GenAI models offer powerful capabilities that can enhance various aspects of application building, from code generation to data processing and understanding natural language. Simultaneously, the microservices architectural pattern has gained prominence for constructing scalable, resilient, and maintainable applications by breaking down complex systems into smaller, independently deployable services. This report explores the synergy between these two powerful paradigms in the context of developing a weather application. The application will be designed to fetch real-time weather data from AccuWeather, a leading weather information provider, utilizing the robust capabilities of Spring Boot for building Java-based microservices. The objective is to provide a comprehensive guide for developing such an application, outlining the key steps involved, from choosing the appropriate GenAI models to designing the microservices architecture and implementing the data fetching and processing logic. This report will delve into the specifics of leveraging GenAI to streamline development and enhance the application's functionality within a Spring Boot microservices environment.

2. Choosing and Integrating GenAI Models with Spring Boot

GenAI models have demonstrated remarkable abilities in various domains, including software development. These models can assist in generating code snippets, parsing complex data structures, and even understanding and responding to natural language queries, thereby significantly boosting developer productivity. The increasing adoption of AI tools by Java developers underscores the growing recognition of their potential to simplify and accelerate the development process.

In the Java and Spring Boot ecosystem, Spring AI stands out as a dedicated framework for building GenAI-powered applications. Spring AI is designed with core principles of portability and modularity, aiming to apply the familiar Spring ecosystem design patterns to the AI domain. It promotes the use of Plain Old Java Objects (POJOs) as fundamental building blocks for AI applications, making it easier for Java developers to integrate AI functionalities into their projects. A significant advantage of Spring AI is its broad support for major AI model providers, including prominent platforms like OpenAI, Microsoft Azure, Amazon Bedrock, Google Vertex AI, Anthropic, and Ollama. It also supports various AI model types, such as Chat Completion for conversational interfaces, Embedding for semantic search, Text to Image for generating visuals, Audio Transcription for converting speech to text, Text to Speech for synthesizing audio from text, and Moderation for content filtering. Furthermore, Spring AI offers seamless integration with major Vector Databases like Neo4j, Chroma, Milvus, and others, which are crucial for implementing Retrieval Augmented Generation (RAG) patterns, enhancing the accuracy and

contextuality of AI model responses. The framework also includes an Advisors API that encapsulates recurring GenAI patterns, simplifying the implementation of common AI tasks. Notably, Spring AI integrates with Spring Initializr, allowing developers to easily bootstrap projects with the necessary AI model and vector store dependencies.
Beyond Spring AI, several other AI tools and platforms can be leveraged for Java and Spring Boot development. Workik AI Code Generator, for instance, offers AI-powered assistance for generating Spring Boot code, including REST APIs and microservices setup. BLACKBOX AI is another coding LLM designed to accelerate software development through features like natural language to code conversion and real-time knowledge integration. Popular AI coding assistants like GitHub Copilot, Codeium, and Tabnine can provide intelligent code completion and suggestions directly within the Integrated Development Environment (IDE), significantly improving coding speed and accuracy. Vertex AI from Google offers a comprehensive suite of machine learning tools that can be integrated with Java applications. These alternative tools can be particularly useful for tasks such as code completion, generating boilerplate code, and identifying potential bugs.
When choosing the right GenAI model for the weather application, several factors should be considered. These include the cost of using the model, its performance in terms of speed and accuracy, the ease of integrating it with Java and Spring Boot, and the specific requirements of the application. For instance, if the application needs to dynamically generate code for interacting with the AccuWeather API or parse complex JSON responses, a model with strong code generation and natural language processing capabilities would be suitable. Given its native integration with the Spring ecosystem and comprehensive support for various AI models, starting with Spring AI appears to be a logical choice for this project.
Integrating Spring AI with a Spring Boot project involves adding the relevant dependencies to the project's build file (e.g., pom.xml for Maven or build.gradle for Gradle). For example, to use OpenAI with Spring AI, the spring-ai-openai-spring-boot-starter dependency can be added. Once the dependency is included, the AI model provider needs to be configured in the application.properties or application.yml file, typically by providing the API key for the chosen provider. After configuration, Spring AI provides convenient interfaces like ChatClient that can be used to interact with the AI model by sending prompts and processing the responses.
The emergence of frameworks like Spring AI signifies a growing trend towards tighter integration of GenAI capabilities within traditional application development environments like Java Spring Boot. This allows developers to leverage the power of large language models and other AI techniques more seamlessly, opening up new possibilities for building intelligent and sophisticated applications.

3. Exploring the AccuWeather API

The weather application will rely on the AccuWeather API to retrieve up-to-date weather information. AccuWeather provides a suite of APIs that offer access to various types of weather data, including forecasts, current conditions, and severe weather alerts. To effectively utilize these APIs, it is crucial to understand the available endpoints, authentication methods, and data formats.
AccuWeather offers a wide range of GET REST endpoints that allow developers to fetch specific weather data. These include the **Locations API**, which is used to obtain a location key necessary for retrieving weather data from other APIs. The **Current Conditions API** provides real-time weather observations for a given location. For forecasting, the **Forecast API** offers

both daily and hourly forecasts, allowing retrieval of predictions for different timeframes. The **Alerts API** delivers information about severe weather warnings issued by official meteorological agencies. Other available endpoints include the **Indices API** for daily index values, the **Weather Alarms API**, the **Imagery API** for radar and satellite images, the **Tropical API** for cyclone information, and the **MinuteCast® API** for short-term precipitation forecasts. Additionally, the API provides access to climatological data through the **Climo API**, tidal forecasts via the **Tidal Forecast API**, and astronomical information through the **Astronomy API**. The **Deep Links API** offers direct links to content on the AccuWeather website and mobile app. For historical data, the **Historical API** allows retrieval of past weather records. The API also includes the **International Air Quality API**, the **River Gauge API**, the **Reports API**, the **Translations API**, and the **Maps API** for various specialized weather-related data. This extensive list of endpoints enables the development of highly customized weather applications tailored to specific data needs.

Accessing the AccuWeather API requires authentication using an API key. There are different tiers of access, including a free developer API with limited access and enterprise APIs for more comprehensive data and higher usage limits. To obtain an API key for the enterprise APIs, it is necessary to contact AccuWeather's sales team. For the free developer API, registration is required on the AccuWeather developer portal. It is essential to keep the API key confidential as it is used to authorize requests and track usage.

The primary data format for responses from the AccuWeather API is JSON (JavaScript Object Notation). JSON is a lightweight and human-readable format that is widely used for data exchange on the web. However, for the Historical API endpoints, AccuWeather also offers the option to receive data in CSV (Comma Separated Values) format, which can be useful for data analysis and processing in tabular form. For detailed information about the specific data formats for each API endpoint, developers should consult the "Data Display Formats" section within the AccuWeather API Reference.

Yes, AccuWeather provides specific endpoints for retrieving current weather conditions and hourly forecasts. The **Current Conditions API** allows fetching the most recent weather information for a given location, including temperature, weather description, wind speed, humidity, and more. Similarly, the **Forecast API** includes an **Hourly Forecasts** endpoint that provides detailed weather predictions for each hour over a specified period. Both of these APIs typically require a location key, which can be obtained using the **Locations API** by providing a geographical identifier like a city name or coordinates.

Below are example JSON responses for current conditions and an hourly forecast, based on the parameters documented by AccuWeather:

Example JSON Response for Current Conditions:


```
{
  "LocalObservationDateTime": "2025-04-06T15:07:00-07:00",
  "EpochTime": 1744101620,
  "WeatherText": "Partly Cloudy",
  "WeatherIcon": 2,
  "HasPrecipitation": false,
  "PrecipitationType": null,
  "IsDayTime": true,
  "Temperature": {
    "Metric": {
      "Value": 20.0,
      "Unit": "C",
```

```
      "UnitType": 17
    },
    "Imperial": {
      "Value": 68.0,
      "Unit": "F",
      "UnitType": 18
    }
  },
  "RealFeelTemperature": {
    "Metric": {
      "Value": 19.4,
      "Unit": "C",
      "UnitType": 17
    },
    "Imperial": {
      "Value": 67.0,
      "Unit": "F",
      "UnitType": 18
    }
  },
  //... other parameters as described in snippet B11
  "MobileLink":
"http://m.accuweather.com/en/us/los-angeles-ca/90001/current-weather/3
47625?lang=en-us",
  "Link":
"http://www.accuweather.com/en/us/los-angeles-ca/90001/weather-forecas
t/347625?lang=en-us"
}
```


Example JSON Response for Hourly Forecast:

Finally, it is crucial to review the AccuWeather API's Terms of Use, which outline the conditions for using the API, including any restrictions on data usage and display. Additionally, developers should be aware of any rate limits associated with their chosen API plan, as exceeding these limits can result in temporary blocking of API access.

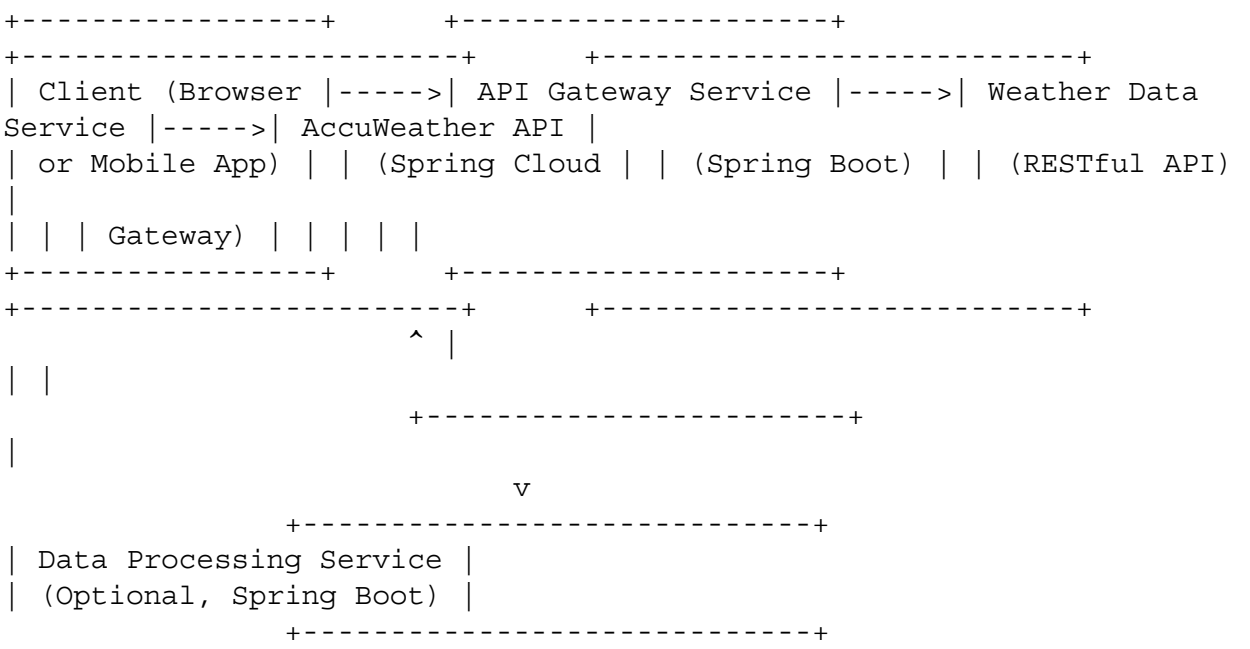
4. Designing the Microservices Architecture for the Weather Application

Adopting a microservices architecture for the weather application offers several advantages. It allows for independent scaling of individual components based on demand, meaning that the service handling API requests can be scaled up during peak times without affecting other parts of the application. Fault isolation is another key benefit; if one microservice experiences a failure, it is less likely to bring down the entire application. Microservices also enable independent deployment, allowing for faster and more frequent updates to specific features without the need to redeploy the entire application. Furthermore, this architecture promotes technology diversity, enabling different teams to choose the most suitable technologies for their

specific service.
A potential microservices architecture for the weather application could comprise the following services:

- **API Gateway:** This service acts as the entry point for all client requests. It handles tasks such as request routing, authentication, and potentially rate limiting. Clients would interact solely with the API Gateway, which would then forward requests to the appropriate backend services.
- **Weather Data Service:** This core service is responsible for interacting directly with the AccuWeather API. It fetches weather data based on location and provides this data to other services within the application. This service encapsulates the logic for making API calls, handling responses, and managing API keys.
- **Data Processing Service (Optional):** This service could be responsible for any necessary data transformation, aggregation, or caching of the weather data retrieved by the Weather Data Service. For instance, it might calculate daily temperature averages or store frequently accessed data in a cache to reduce the load on the AccuWeather API.
- **User Interface Service (Out of Scope for Initial Request):** While the initial request focuses on data fetching, a complete application would likely include a user interface. This service would be responsible for rendering the UI and displaying the weather information to the user.

A simplified architecture diagram would look like this:



For the backend microservices, particularly the Weather Data Service and the optional Data Processing Service, Spring Boot is an excellent choice due to its ease of use and extensive ecosystem. The API Gateway could be implemented using Spring Cloud Gateway, which provides powerful routing and filtering capabilities. For the User Interface Service, various front-end technologies like React, Angular, or Vue.js could be used, depending on the specific requirements.
Communication between the microservices can be facilitated using RESTful APIs with JSON as the data format. For example, the API Gateway would make REST calls to the Weather Data Service to retrieve weather information. In scenarios where more decoupled or asynchronous

communication is needed, message queues like RabbitMQ or Kafka could be employed.
To manage the distributed nature of the microservices, service discovery mechanisms like Spring Cloud Netflix Eureka can be used. This allows services to dynamically discover and communicate with each other without hardcoding network locations. Centralized configuration management tools like Spring Cloud Config can also be beneficial for managing the configuration of all the microservices in a consistent and scalable manner.
If the application requires storing processed or cached weather data, various database options can be considered. For relational data, PostgreSQL or MySQL could be used. For NoSQL needs, MongoDB might be a suitable choice. In-memory data stores like Redis are well-suited for caching frequently accessed data.
By adopting a microservices architecture, the weather application can be built in a modular and scalable way, allowing for future enhancements and easier maintenance.

5. Setting Up the Spring Boot Project

To begin building the weather application, it is essential to set up the Spring Boot project structure correctly for each microservice. A typical and recommended structure for a Spring Boot REST API project includes organizing the code into logical packages. For the **Weather Data Service** microservice, a structure like the following can be adopted:

```
src/
├── main/
│   ├── java/
│   │   └── com/example/weatherapp/
│   │       ├── config/          // Configuration classes
│   │       ├── controller/      // REST controllers
│   │       ├── model/           // Data model classes (for
AccuWeather API responses)
│   │       ├── service/         // Business logic services
│   │       ├── client/          // Client for AccuWeather API
interaction
│   │       ├── dto/             // Data transfer objects (internal
representation)
│   │       ├── exception/       // Custom exception classes
│   │       └── WeatherAppApplication.java // Main application class
│   └── resources/
│       ├── application.properties  // Application configuration
│       └── log4j2.xml             // Logging configuration (optional)
└── test/
    └── java/
        └── com/example/weatherapp/
            ├── controller/
            └── service/
```

Similarly, for the **API Gateway** microservice (if using Spring Cloud Gateway), a similar structure can be used, focusing on routing configurations and potentially authentication and authorization logic.
The next step is to include the necessary dependencies in the project's pom.xml (for Maven) or

build.gradle (for Gradle) file. For the **Weather Data Service** microservice, the following essential dependencies should be included:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-json</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.experimental.ai</groupId>
    <artifactId>spring-ai-openai-spring-boot-starter</artifactId>
    <version>0.8.0-SNAPSHOT</version> </dependency>
```

For Gradle:

```
// Gradle
implementation 'org.springframework.boot:spring-boot-starter-web'
implementation 'org.springframework.boot:spring-boot-starter-webflux'
implementation 'org.springframework.boot:spring-boot-starter-json'
implementation 'org.springframework.boot:spring-boot-starter-log4j2'
implementation 'org.springframework.boot:spring-boot-starter-actuator'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
// Spring AI dependency (choose specific starter based on LLM)
implementation
'org.springframework.experimental.ai:spring-ai-openai-spring-boot-star
ter:0.8.0-SNAPSHOT' // Use the latest version
```

The spring-boot-starter-web dependency is essential for building RESTful APIs using Spring MVC, while spring-boot-starter-webflux is used for the non-blocking REST client, WebClient. spring-boot-starter-json provides support for JSON processing. spring-boot-starter-log4j2 is for

logging, spring-boot-starter-actuator enables monitoring capabilities , and spring-boot-starter-test is for writing unit and integration tests. The spring-ai-openai-spring-boot-starter dependency (or a similar starter for other LLMs) is needed for integrating Spring AI with the chosen GenAI model.
For the **API Gateway** microservice (if implemented using Spring Cloud Gateway), the following dependencies would be crucial:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

For Gradle:

```
// Gradle
implementation 'org.springframework.boot:spring-boot-starter-webflux'
implementation
'org.springframework.cloud:spring-cloud-starter-gateway'
// If using Eureka for service discovery
implementation
'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client'
```

These dependencies provide the necessary libraries for building a reactive web application with Spring Cloud Gateway and for registering with a Eureka server for service discovery, if that approach is chosen.
The choice between using spring-boot-starter-web with RestTemplate or RestClient, and spring-boot-starter-webflux with WebClient, depends on whether a blocking or non-blocking approach is preferred for making HTTP requests. WebClient, being non-blocking, is generally recommended for better performance and scalability in microservices environments. Spring Boot 3.2 introduced RestClient as a modern, fluent API alternative built on top of WebClient.
By setting up the correct project structure and including the necessary dependencies, the foundation for building the Spring Boot microservices for the weather application is established.

6. Implementing Weather Data Fetching using Spring Boot

The Weather Data Service will be responsible for fetching weather data from the AccuWeather API. This involves creating a REST client, making requests to the appropriate API endpoints, and handling the responses.
A Spring RestClient or WebClient bean can be created to interact with the AccuWeather API.

Using the RestClient.builder() or WebClient.builder() method allows for configuring the base URL of the AccuWeather API, which would be apidev.accuweather.com for development purposes. The API key obtained from AccuWeather needs to be included in every request, typically as a query parameter named apikey or through a custom header, as specified by the AccuWeather API documentation.
To fetch current weather conditions, a GET request needs to be made to the Current Conditions API endpoint. This usually requires a location key, which can be obtained by first calling the Locations API with a city name or geographical coordinates. The response from the Locations API will contain the location key, which can then be used in the Current Conditions API request. The RestClient or WebClient can be used to make this GET request, specifying the endpoint URL and including the API key and location key as parameters. The response will be in JSON format and needs to be handled appropriately, potentially using Java data models defined in the next section. Error handling should be implemented to manage potential issues like network errors or invalid API responses, possibly using try-catch blocks or WebClient's reactive error handling capabilities.
Similarly, to fetch hourly forecasts, a GET request is made to the Hourly Forecasts API endpoint, again requiring a location key. The response will be a JSON array of forecast entries, each representing the weather conditions for a specific hour. This array needs to be processed, and the data can be mapped to appropriate Java objects.
When handling API responses, Spring's JSON processing capabilities, facilitated by the @ResponseBody annotation and the ObjectMapper, can be used to automatically convert the JSON response to instances of Java classes. It is important to ensure that the structure of the Java data models matches the structure of the JSON responses from the AccuWeather API to enable correct deserialization. In cases where the response format might deviate from the expected structure or encoding, it might be necessary to configure custom message converters or handle the parsing manually.
The AccuWeather API's requirement for a location key before fetching weather data necessitates a two-step process. First, the location key must be obtained based on a geographical identifier. Then, this key is used to retrieve the actual weather information. This design likely allows AccuWeather to efficiently manage and serve location-specific weather data. Implementing robust error handling is crucial when interacting with external APIs, as various issues such as network connectivity problems, API rate limits being exceeded, or invalid requests can lead to failures. Handling these errors gracefully ensures the application's stability and provides a better user experience. Finally, a thorough understanding of the JSON responses from the AccuWeather API is essential for correctly mapping the data to Java objects. Tools for viewing and inspecting JSON structures can be very helpful in this process.

7. Defining Java Data Models for Weather Information

To effectively process the JSON responses from the AccuWeather API, it is necessary to define corresponding Java classes (POJOs) that will represent the weather data. These classes should have fields that map to the JSON attributes in the API responses.
For the Current Conditions API, based on the parameters described in snippet B11, the following Java classes can be created:

- **CurrentConditions:** This class will be the main class representing the current weather conditions. It will contain fields like localObservationDateTime, epochTime, weatherText, weatherIcon, temperature (referencing a Temperature object), realFeelTemperature

(referencing a RealFeelTemperature object), relativeHumidity, wind (referencing a Wind object), pressure (referencing a Pressure object), and others. The @JsonProperty annotation from the Jackson library should be used to map the JSON field names to the Java property names.
- **Temperature:** This class will represent the temperature in both metric and imperial units. It will have fields for metric (referencing a UnitValue object for Celsius) and imperial (referencing a UnitValue object for Fahrenheit).
- **RealFeelTemperature:** Similar to Temperature, this class will represent the RealFeel™ temperature in both metric and imperial units, also including a descriptive phrase.
- **UnitValue:** This class will represent a value with its unit (e.g., 20.0 Celsius). It will have fields for value, unit, and unitType.
- **Wind:** This class will contain information about the wind, including speed (referencing a UnitValue object), and direction (with fields for degrees, localized abbreviation, and English abbreviation).
- **Pressure:** This class will represent the pressure in both metric (mb) and imperial (inHg) units, using UnitValue objects.

Similarly, for the Hourly Forecasts API, based on the parameters described in snippet B14, the following Java classes can be created:

- **HourlyForecast:** This class will represent a single hour's forecast. It will contain fields like dateTime, epochDateTime, weatherIcon, iconPhrase, temperature (referencing a Temperature object), wind (referencing a Wind object), precipitationProbability, and others. Again, @JsonProperty should be used for mapping.
- **Temperature:** This class can be reused from the Current Conditions data model.
- **Wind:** This class can also be reused.
- **Precipitation:** A class to represent precipitation details (e.g., type, intensity, amount).

When designing these data models, it is important to closely mirror the structure of the JSON responses from the AccuWeather API to ensure seamless deserialization. Tools that can generate Java classes from JSON schemas can be particularly helpful in this process. Furthermore, considering potential future needs and designing the data models in a way that allows for easy addition of new fields or support for other AccuWeather API endpoints is advisable. For instance, using nested classes to represent complex data structures in the JSON response can improve the organization and readability of the Java code.

Adopting immutable data objects, where fields are final and only getter methods are provided, can enhance the robustness and thread safety of the application, especially in a microservices environment. This prevents unintended modifications to the data once it is created. While the initial design should focus on the current requirements, anticipating potential future needs for additional weather data or features is important. Designing for extensibility from the beginning can save significant effort in the long run when the application needs to be adapted to new requirements.

8. Leveraging GenAI for Code Generation in the Weather Application

GenAI models can be a powerful tool in accelerating the development of the weather application. They can be used for various code generation tasks, including creating code snippets for API calls, parsing JSON responses, and even generating basic service logic.

Here are some example prompts that could be used with a GenAI model (like those supported

by Spring AI) to generate code for this project:

- **Prompt for API Call:** "Generate a Spring RestClient method in Java to make a GET request to the AccuWeather Current Conditions API for location key 'XXXX' including the API key 'YYYY' as a query parameter."
- **Prompt for Data Parsing:** "Generate Java code using Jackson annotations to map the following JSON response from the AccuWeather Current Conditions API to a Java class named CurrentConditions: {\"LocalObservationDateTime\":\"2025-04-06T15:07:00-07:00\",\"EpochTime\":1744101620,\"WeatherText\":\"Partly Cloudy\",...}."
- **Prompt for Service Logic:** "Generate a Spring Boot service method in Java that takes a list of HourlyForecast objects and returns a list of only those forecasts where the precipitation probability is greater than 50%."
- **Prompt for Unit Test:** "Generate a JUnit test case for the WeatherService class in a Spring Boot application. This test should mock the AccuWeatherClient and verify that the getCurrentConditions method correctly handles a successful API response by checking the WeatherText field."

Once the GenAI model generates the code snippets based on these prompts, the generated code can be copied and integrated into the Spring Boot project. However, it is crucial to review and test the generated code thoroughly to ensure its correctness, security, and adherence to the project's coding standards. GenAI can significantly speed up the development process by automating the creation of repetitive code, allowing developers to focus on more complex aspects of the application. However, the quality and relevance of the generated code heavily depend on the clarity and specificity of the prompts provided to the AI model. Effective prompt engineering is therefore an important skill when leveraging GenAI for code generation. While GenAI is a valuable tool, it should not be considered a complete replacement for human developers. The generated code might require adjustments, optimizations, or the addition of error handling logic that the AI model might have overlooked. Human oversight remains essential to ensure the overall quality and reliability of the application.

9. Implementing Robust Error Handling, Logging, and Monitoring in Microservices

For the weather application's microservices to be robust and maintainable, it is crucial to implement effective error handling, logging, and monitoring mechanisms.

In Spring Boot, error handling can be implemented using various approaches. The @ExceptionHandler annotation can be used within a controller to handle specific exceptions and return custom error responses to the client, often using the ResponseEntity class. For more global error handling across the application, a @ControllerAdvice class can be created to intercept exceptions and provide consistent error responses. When making API calls using RestClient or WebClient, it is important to handle potential exceptions like HttpClientErrorException (for 4xx status codes) and HttpServerErrorException (for 5xx status codes) to gracefully manage issues arising from the AccuWeather API.

Logging is essential for tracking the behavior of the microservices, diagnosing issues, and gaining insights into the application's runtime. Spring Boot supports several logging frameworks, with Log4j2 being a popular choice (as indicated by its inclusion in the dependencies). Logging should include relevant information such as request details, responses from the AccuWeather API, any errors encountered, and significant application events. Using structured logging

formats (e.g., JSON) can make it easier to analyze logs using centralized logging systems. Spring Boot Actuator provides built-in endpoints for monitoring the health, metrics, and other operational aspects of the microservices. By including the spring-boot-starter-actuator dependency, endpoints like /health (to check the application's health status), /metrics (to expose various application metrics), and /info (to display application information) become available. These endpoints can be enabled and configured in the application.properties or application.yml file. For more advanced monitoring, Actuator can be integrated with external monitoring tools like Prometheus and Grafana, or commercial solutions like Dynatrace and AppDynamics. Implementing comprehensive error handling, logging, and monitoring is vital for the operational stability and maintainability of the weather application's microservices. These practices enable developers to quickly identify and resolve issues, understand the application's performance, and ensure a reliable service. Standardized error response formats across all microservices improve consistency and make it easier for clients to handle errors. Spring Boot Actuator offers a straightforward way to implement basic monitoring with minimal configuration, providing immediate visibility into the health and performance of the microservices.

10. Exploring Advanced Data Processing and Potential Integrations

Beyond the basic functionality of fetching and displaying weather data, several advanced data processing techniques and potential integrations can enhance the weather application. Caching frequently accessed weather data can significantly reduce the load on the AccuWeather API and improve the application's response time. Spring provides a caching abstraction that can be used with various caching providers like Redis or Caffeine. By caching the responses from the AccuWeather API for a certain duration, the application can serve subsequent requests for the same data from the cache, avoiding unnecessary API calls and improving performance.

In some scenarios, the raw weather data fetched from AccuWeather might need further processing or aggregation before it can be used by other services or displayed to the user. For example, the application might need to calculate the average temperature for the day based on hourly forecasts or combine data from different AccuWeather API endpoints to provide a more comprehensive view. A dedicated Data Processing Service (as discussed in the microservices architecture) can handle such tasks.

The Weather Data Service can be integrated with other microservices or applications to create more sophisticated functionalities. For instance, it could be integrated with a notification service to send users weather alerts based on specific conditions or with a smart home application to automatically adjust thermostats based on the weather forecast. Such integrations can be achieved through well-defined APIs or by using message queues for asynchronous communication.

For applications that require real-time weather updates, technologies like WebSockets or Server-Sent Events (SSE) can be used to push data from the server to the clients as it becomes available. This can provide a more dynamic and engaging user experience compared to traditional request-response patterns.

Caching is indeed a critical optimization technique for applications consuming external APIs, as it reduces latency and costs associated with repeated calls. By storing frequently requested weather data, the application can serve these requests quickly without hitting the AccuWeather API each time. The weather data obtained from AccuWeather is valuable for various other

applications and services, allowing for the development of more integrated solutions. For example, weather data could be used in agricultural applications for optimizing irrigation schedules or in transportation services for predicting travel conditions. Real-time data streaming can significantly improve the user experience by providing immediate weather updates without manual refreshes, making the application more interactive and informative.

11. Building and Running the Spring Boot Microservices Locally

To test and develop the weather application, it is essential to be able to build and run the Spring Boot microservices locally.
First, ensure that the Java Development Kit (JDK) is installed on your system, along with a suitable Integrated Development Environment (IDE) like IntelliJ IDEA or Eclipse. Additionally, Maven or Gradle should be installed based on the project's build configuration.
To build the Spring Boot microservices, navigate to the root directory of each service in your terminal and run the appropriate build command. For Maven projects, use mvn clean package. This command compiles the code, runs tests, and packages the application as an executable JAR file in the target directory. For Gradle projects, use ./gradlew build. This performs similar steps and creates the JAR file in the build/libs directory.
Once the JAR files are created, each microservice can be run locally. You can do this directly from your IDE by running the main application class (e.g., WeatherAppApplication.java). Alternatively, you can run the JAR files from the command line using the java -jar <service-name>.jar command, where <service-name>.jar is the name of the generated JAR file. Spring Boot applications typically run on port 8080 by default. If you are running multiple microservices locally, you will need to configure different ports for each service in their respective application.properties or application.yml files by setting the server.port property.
After running the microservices, you can test their API endpoints using tools like Postman, Insomnia, or curl. For the Weather Data Service, you would typically send requests to endpoints defined in the controller classes to fetch current conditions or hourly forecasts for a specific location. For example, you might send a GET request to http://localhost:8080/weather/current?location=London (assuming an endpoint like this is defined).
For a more consistent and isolated development environment, consider using Docker and Docker Compose to containerize and run the microservices locally. This involves creating a Dockerfile for each service that specifies the environment and dependencies, and a docker-compose.yml file to define and manage the multi-container setup. Here is a basic example of a Dockerfile for the Weather Data Service:

```
FROM openjdk:17-jdk-slim
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

And a basic docker-compose.yml file:

```
version: '3.8'
services:
  weather-data-service:
```

```
    build:./weather-data-service
    ports:
      - "8080:8080"
    environment:
      - ACCUWEATHER_API_KEY=your_api_key
```

To use Docker, you would first build the Docker image for each service using docker build -t weather-data-service./weather-data-service (assuming the Dockerfile is in the weather-data-service directory). Then, you can run all the services defined in the docker-compose.yml file using the command docker-compose up. Using Docker ensures that each microservice runs in a consistent environment, reducing potential issues related to different operating systems or software configurations on development machines. Thorough local testing of each microservice, including unit tests, integration tests, and manual testing of the API endpoints, is crucial before deploying to a production environment. This helps in identifying and fixing bugs early in the development cycle.

12. Conclusion

Building a weather application that fetches data from AccuWeather using GenAI, Spring Boot, and microservices involves several key steps. These include selecting and integrating appropriate GenAI models using frameworks like Spring AI, exploring the AccuWeather API to understand available endpoints and authentication, designing a robust microservices architecture, setting up the Spring Boot projects with necessary dependencies, implementing the logic for fetching and handling weather data, defining Java data models to represent the API responses, leveraging GenAI for code generation to accelerate development, and implementing proper error handling, logging, and monitoring for operational stability. Additionally, considering advanced data processing techniques like caching and exploring potential integrations with other services can further enhance the application's value. Finally, the ability to build and run the microservices locally in a consistent environment is crucial for effective development and testing.

The combination of Spring Boot's ease of use for building microservices, the power of GenAI for streamlining development and adding intelligent features, and the rich data provided by the AccuWeather API offers a compelling approach to creating sophisticated weather applications. The microservices architecture ensures scalability and maintainability, while GenAI can assist in various development tasks, making the process more efficient. Potential future enhancements could include developing a user-friendly interface, incorporating more advanced data analysis and visualization, integrating with other weather data sources, or deploying the application to a cloud platform for broader accessibility. While building such an application presents certain challenges, the benefits of leveraging these modern technologies make it a worthwhile endeavor for creating intelligent and scalable solutions.

Works cited

1. Large Language Models for Software Engineering: Survey and Open Problems - COINSE, https://coinse.github.io/publications/pdfs/Fan2023yu.pdf 2. A real-time and modular weather station software architecture based on microservices - ADS, https://ui.adsabs.harvard.edu/abs/2025EnvMS.18606337B/abstract 3. Weather Forecast

Microservice Application Overview - Online Documentation Platform, https://doc.hcs.huawei.com/usermanual/servicestage/servicestage_05_0002.html 4. Best Large Language Models for Java - SourceForge, https://sourceforge.net/software/large-language-models/integrates-with-java/ 5. AI Tools Now Essential in Java Dev's Productivity Arsenal - The New Stack, https://thenewstack.io/ai-tools-now-essential-in-java-devs-productivity-arsenal/ 6. GenAI Starter Kit: Everything You Need to Build an Application with Spring AI in Java - Neo4j, https://neo4j.com/blog/developer/genai-starter-kit-spring-java/ 7. Spring AI, https://spring.io/projects/spring-ai/ 8. Building a Generative AI Application with Spring AI | by Bradley Clemson | Version 1, https://medium.com/version-1/building-a-generative-ai-application-with-spring-ai-dce717e38526 9. Spring Boot AI: A Comprehensive Guide to Intelligent Java Development, https://master-spring-ter.medium.com/spring-boot-ai-a-comprehensive-guide-to-intelligent-java-development-c5bdf7116a50 10. Integrating AI with Spring Boot: A Beginner's Guide - mydeveloperplanet.com, https://mydeveloperplanet.com/2025/01/08/integrating-ai-with-spring-boot-a-beginners-guide/ 11. spring.io, https://spring.io/projects/spring-ai#:~:text=Features,Amazon%2C%20Google%2C%20and%20Ollama. 12. Spring AI Concepts Tutorial With Examples - Chat Model API - JavaTechOnline, https://javatechonline.com/spring-ai-concepts-tutorial-with-examples/ 13. Spring AI Framework Overview – Introduction to AI World for Java Developers - Grape Up, https://grapeup.com/blog/spring-ai-overview-introduction-to-ai-world-for-java-developers/ 14. Introduction :: Spring AI Reference, https://docs.spring.io/spring-ai/reference/ 15. spring-projects/spring-ai: An Application Framework for AI Engineering - GitHub, https://github.com/spring-projects/spring-ai 16. Free AI-powered Spring Boot Code Generator: Simplify Java Development - Workik, https://workik.com/spring-boot-code-generator 17. Top 13 AI Tools For Java Developers To Boost Productivity - Groove Technology, https://groovetechnology.com/blog/technologies/ai-for-java-developers/ 18. 15 Best AI Coding Assistant Tools in 2025 - Qodo, https://www.qodo.ai/blog/best-ai-coding-assistant-tools/ 19. Best AI Tools for Java - SourceForge, https://sourceforge.net/software/ai-tools/integrates-with-java/ 20. What is thd best AI-powered code editor? - java - Reddit, https://www.reddit.com/r/java/comments/1ijx952/what_is_thd_best_aipowered_code_editor/ 21. AccuWeather Enterprise API Documentation, http://apidev.accuweather.com/ 22. API Reference - AccuWeather APIs, https://developer.accuweather.com/apis 23. Frequently Asked Questions - AccuWeather APIs, https://developer.accuweather.com/faq-page 24. AccuWeather API Data Service - Microsoft AppSource, https://appsource.microsoft.com/en-gb/product/web-apps/accuweather.accuweather_forecast_api?tab=Overview 25. How to Build a REST API using Java Spring Boot - Index.dev, https://www.index.dev/blog/build-rest-api-java-spring-boot 26. Learn Project structure | Spring Boot Basics - Codefinity, https://codefinity.com/courses/v2/87dc501e-89a6-4a4b-afd4-8b38c46a80c7/9a2cd386-7414-4da6-a5ba-c79732024d97/c5672c70-6af2-4164-ad97-b6205befa155 27. adityamputra27/weather-app-microservice - GitHub, https://github.com/adityamputra27/weather-app-microservice 28. Simple microservice which handles weather data - GitHub, https://github.com/avedmala/weather-microservice 29. What is the recommended project structure for spring boot rest projects? - Stack Overflow, https://stackoverflow.com/questions/40902280/what-is-the-recommended-project-structure-for-spring-boot-rest-projects 30. The recommended Spring Boot project structure leads to repetitive

code,
https://softwareengineering.stackexchange.com/questions/445668/the-recommended-spring-boot-project-structure-leads-to-repetitive-code 31. How to structure my REST API : r/SpringBoot - Reddit, https://www.reddit.com/r/SpringBoot/comments/1am0fds/how_to_structure_my_rest_api/ 32. Spring WebClient vs RestTemplate: What's better in 2024? | by Pushkar Kumar | Medium, https://medium.com/@kmrpushkar09/spring-webclient-vs-resttemplate-whats-better-in-2023-99844f649b53 33. RestClient vs. WebClient vs RestTemplate: Choosing the right library to call REST API in Spring Boot - Digma AI, https://digma.ai/restclient-vs-webclient-vs-resttemplate/ 34. Is there still a need for webflux? : r/java - Reddit, https://www.reddit.com/r/java/comments/1aldwzg/is_there_still_a_need_for_webflux/ 35. WebClient vs RestTemplate vs FeignClient: A Comparative Guide - DEV Community, https://dev.to/nullvoidkage/webclient-vs-resttemplate-vs-feignclient-a-comparative-guide-4028 36. Spring Boot 3.2: Replace Your RestTemplate With RestClient - DZone, https://dzone.com/articles/spring-boot-32-replace-your-resttemplate-with-rest 37. Spring Framework – Internals of RestClient - Foojay.io, https://foojay.io/today/spring-internals-of-restclient/ 38. A Guide to RestClient in Spring Boot - Baeldung, https://www.baeldung.com/spring-boot-restclient 39. REST Clients :: Spring Framework, https://docs.spring.io/spring-framework/reference/integration/rest-clients.html 40. Deserialize text json with Rest Client in Java Spring Boot - Stack Overflow, https://stackoverflow.com/questions/78021722/deserialize-text-json-with-rest-client-in-java-spring-boot 41. How to handle in spring boot with rest client - json - Stack Overflow, https://stackoverflow.com/questions/78176865/how-to-handle-in-spring-boot-with-rest-client 42. How do you do Orchestration and Design in Java Microservices? - Aegis Softtech, https://www.aegissofttech.com/articles/orchestration-and-design-in-java-microservices.html 43. Weather Data APIs: Real-time & historical weather intelligence, https://www.weathercompany.com/weather-data-apis/