Creating a website for providing itineraries for travel using AWS and implementing a three-tier architecture is a great idea. Here's a basic outline of how you could approach this:

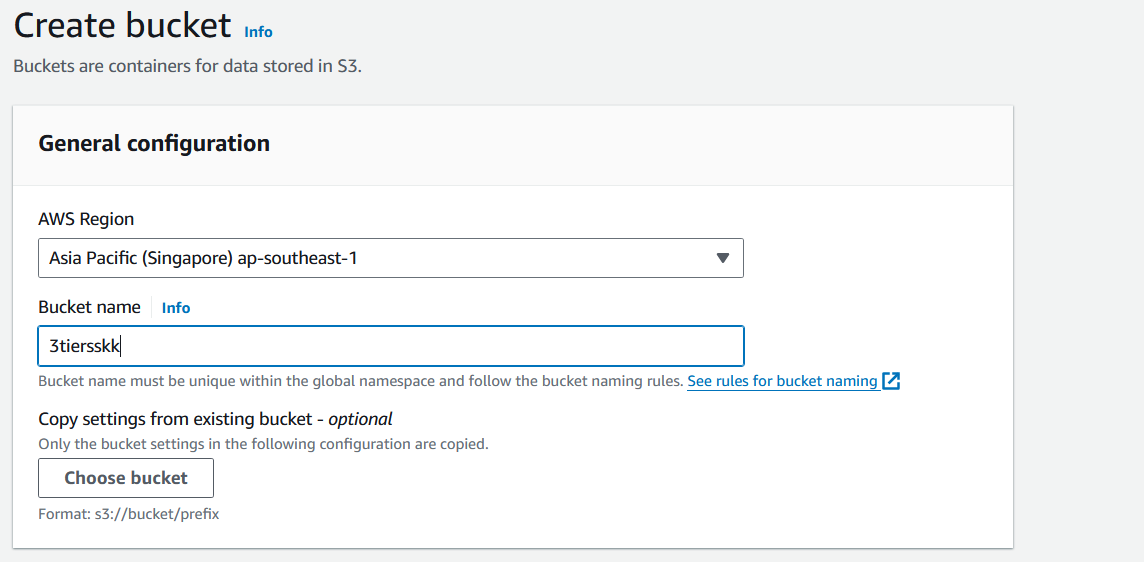
1. **Frontend**: This is the user interface layer where users interact with your website.
2. **Backend**: This layer handles the logic of your application, including processing user requests, fetching data, and generating responses.
3. **Database**: This layer stores and retrieves data needed for your application.

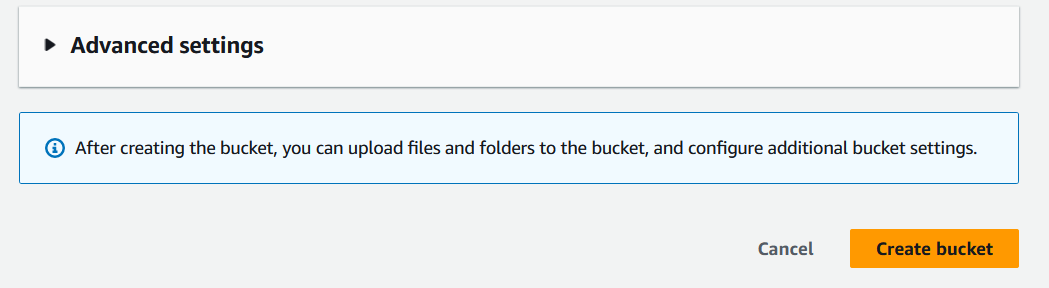
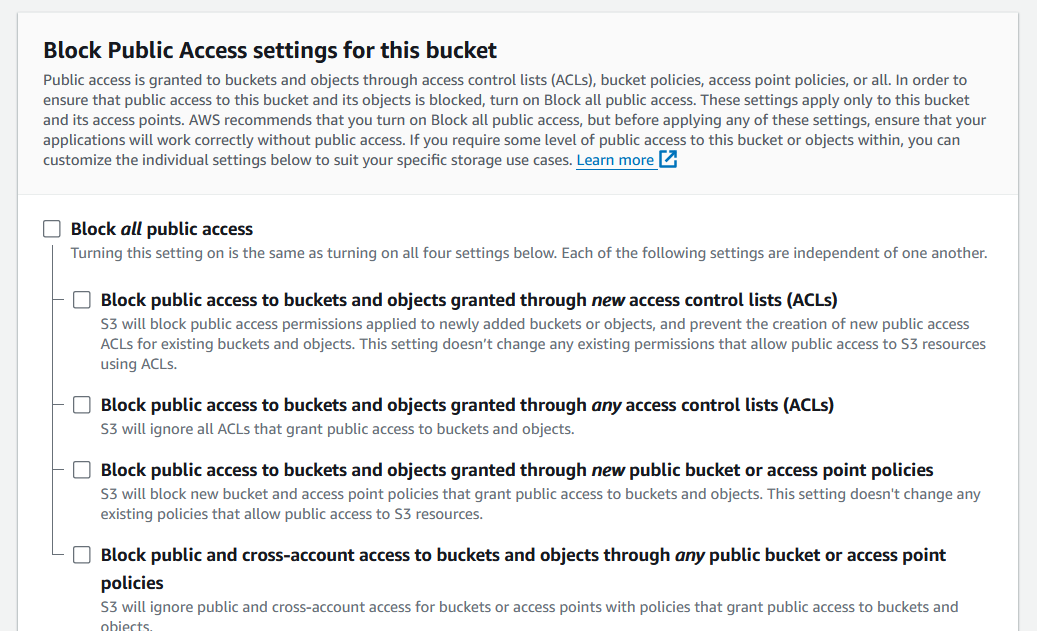
Here's how you can implement each tier using AWS services:

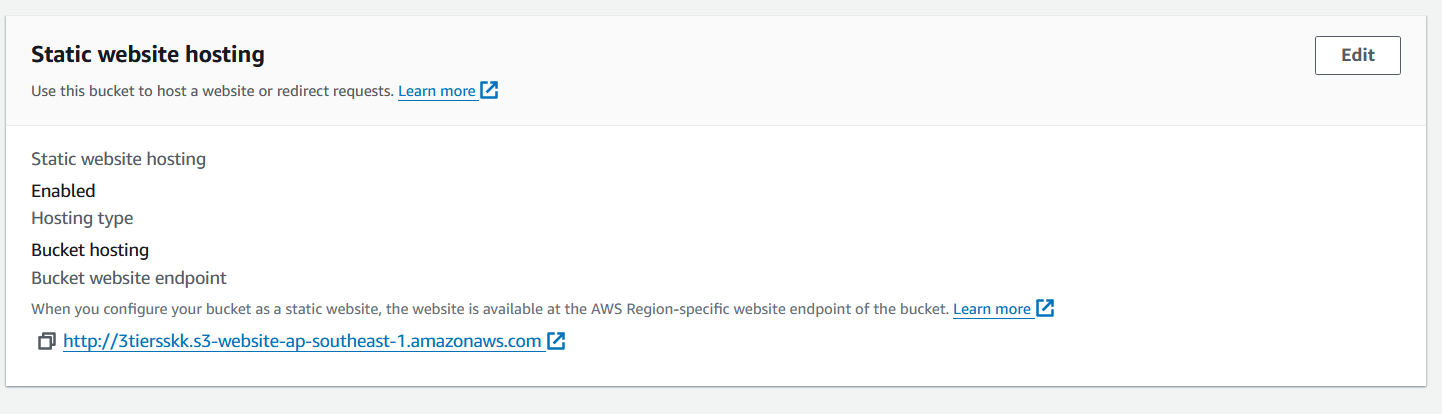
**Frontend (Presentation Tier):**

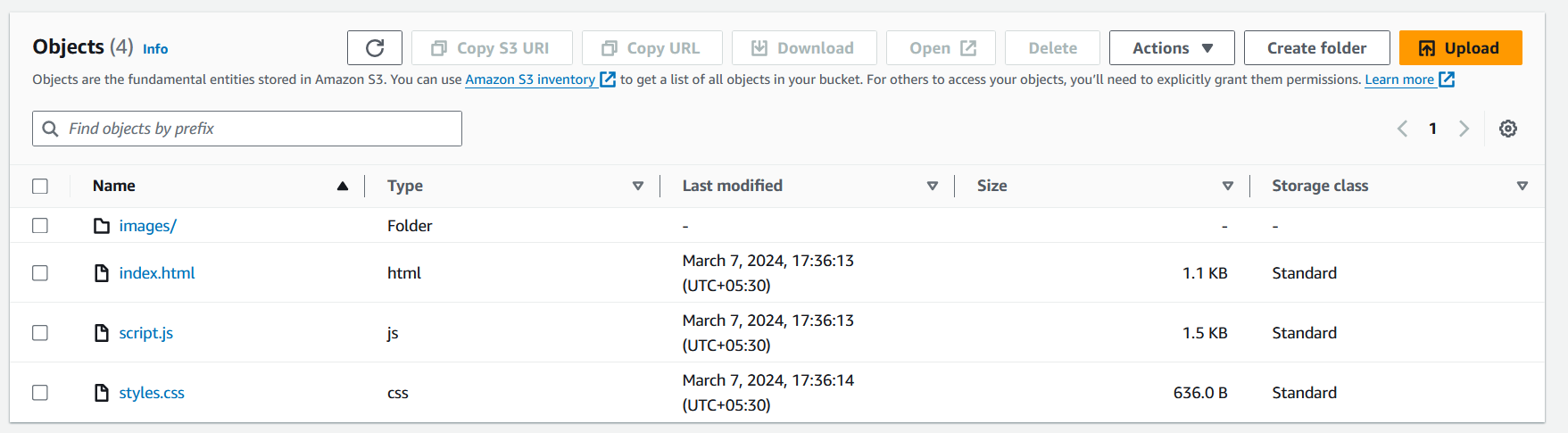
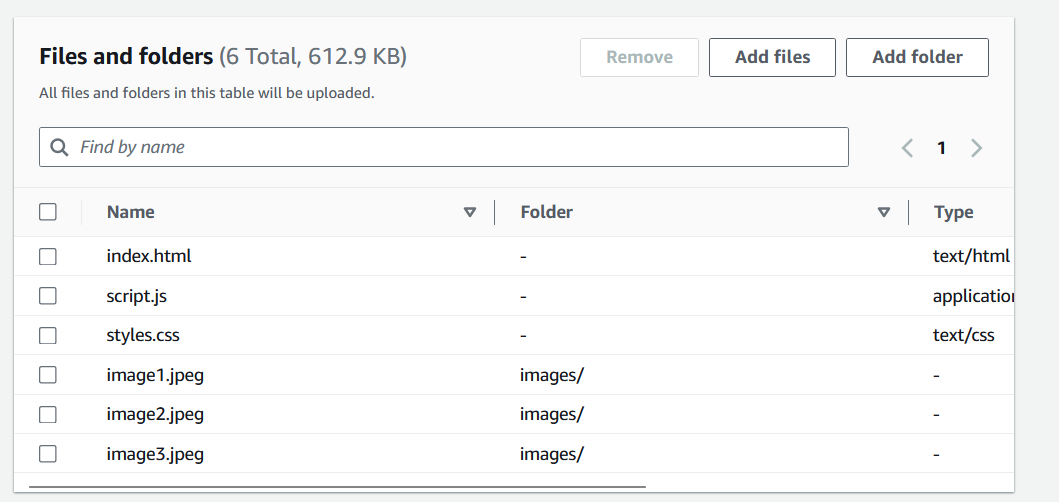
For the frontend, you can use AWS services like:

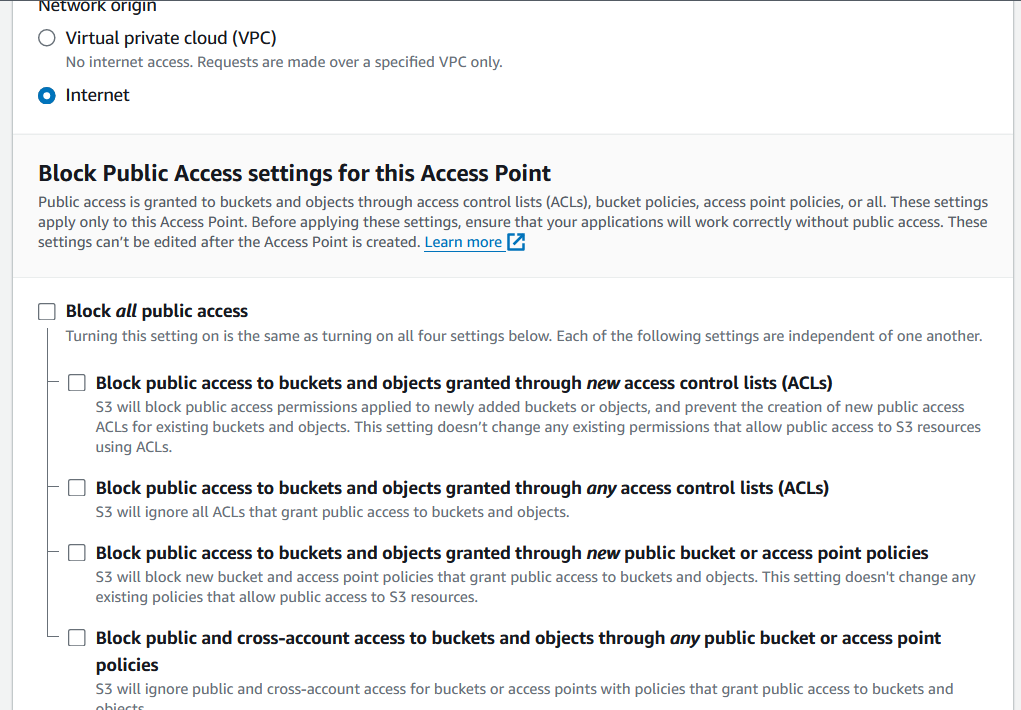
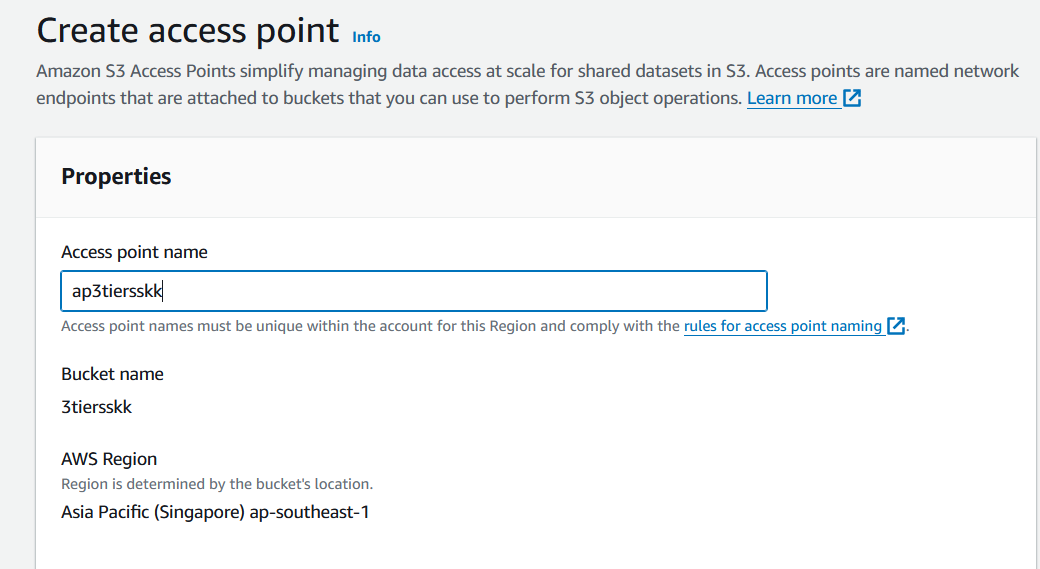
* **Amazon S3 (Simple Storage Service)**: Host your static website files (HTML, CSS, JavaScript, images) on S3.

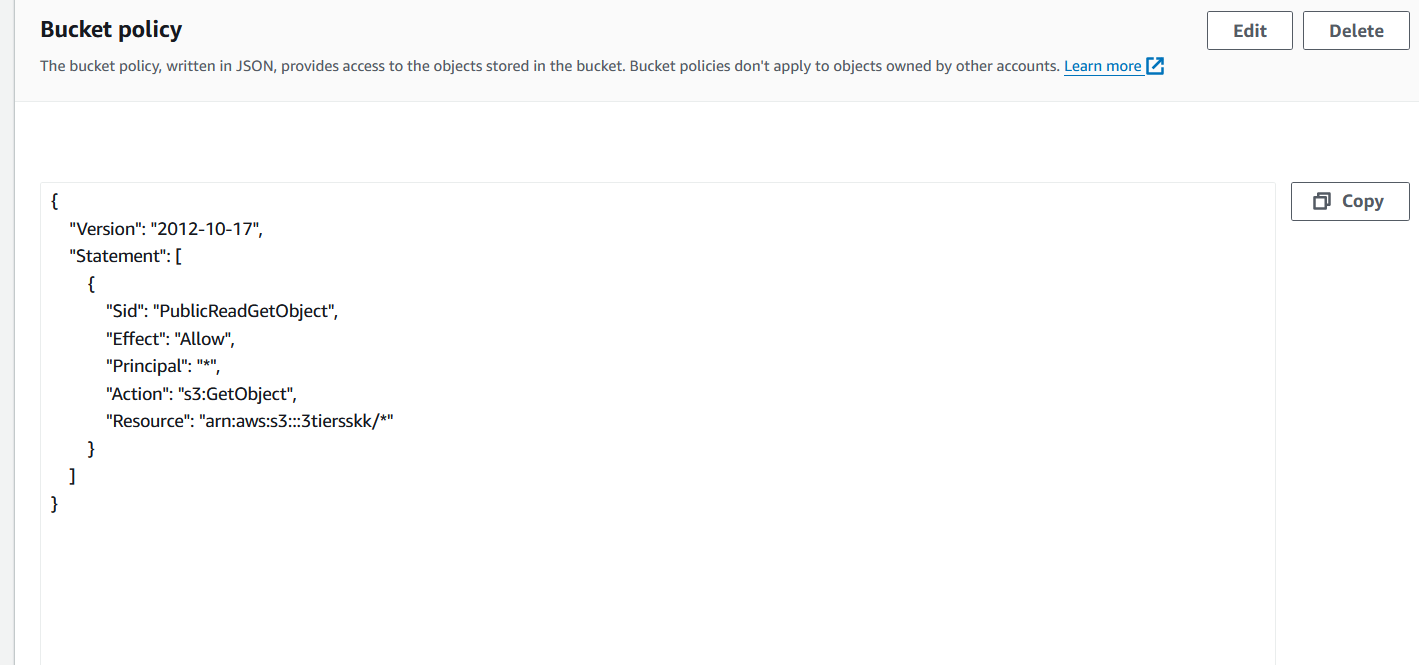




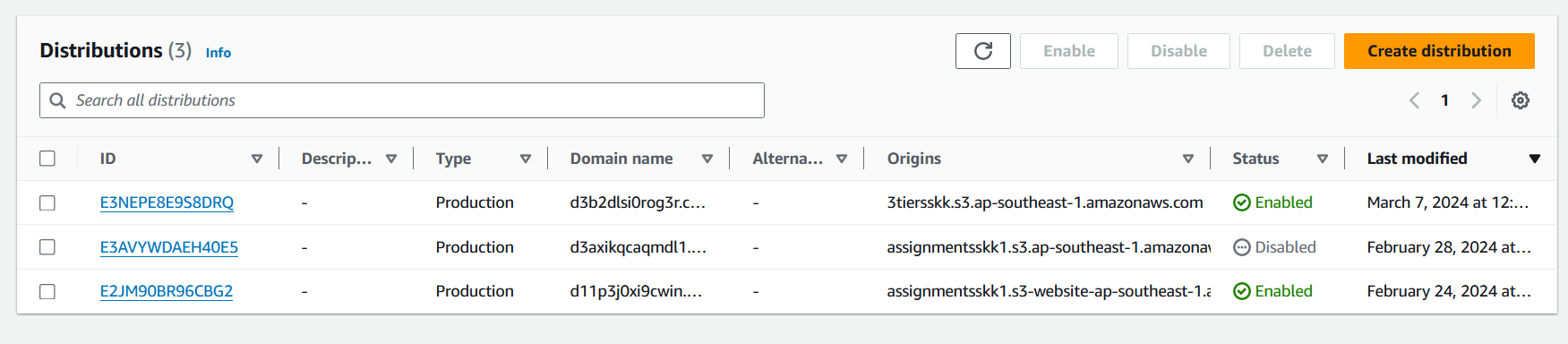
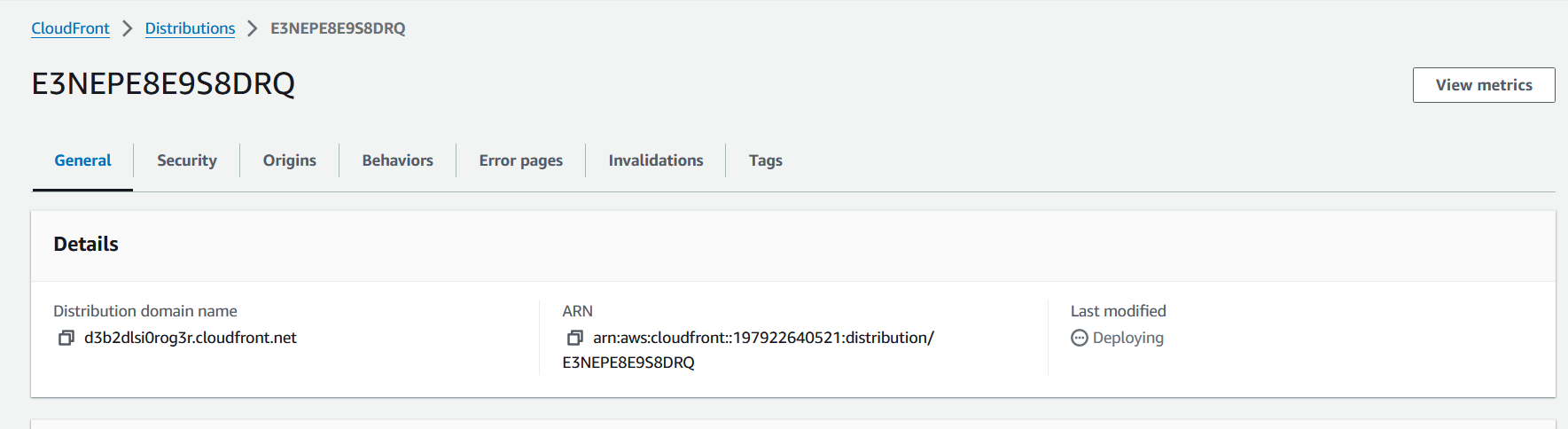
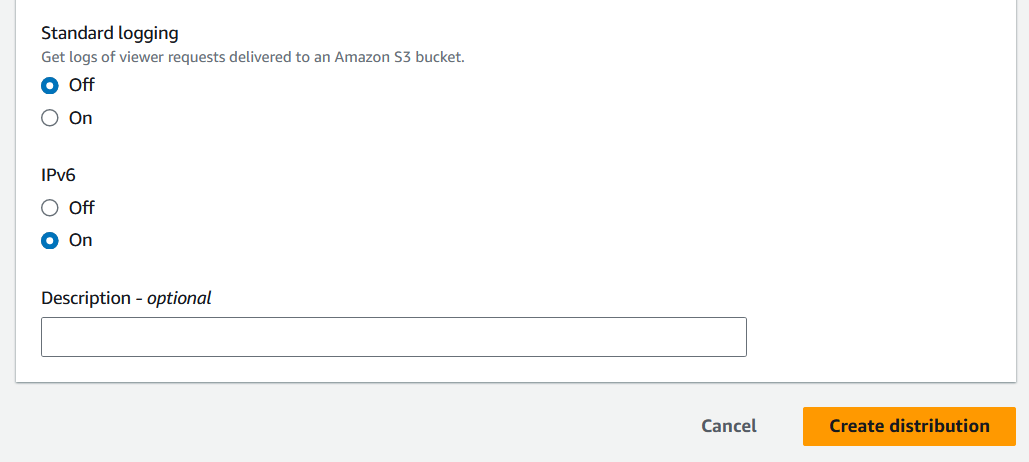
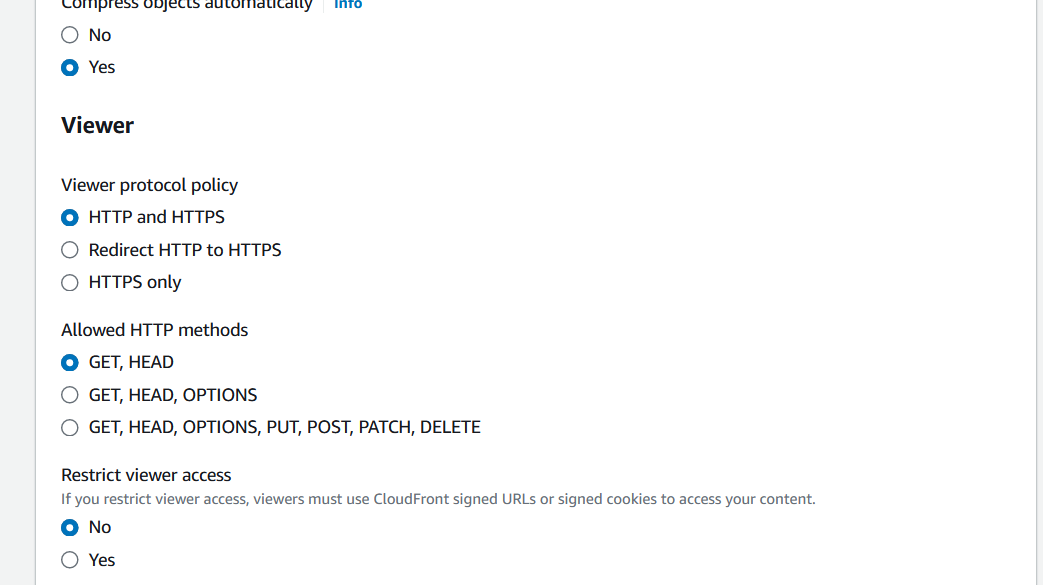
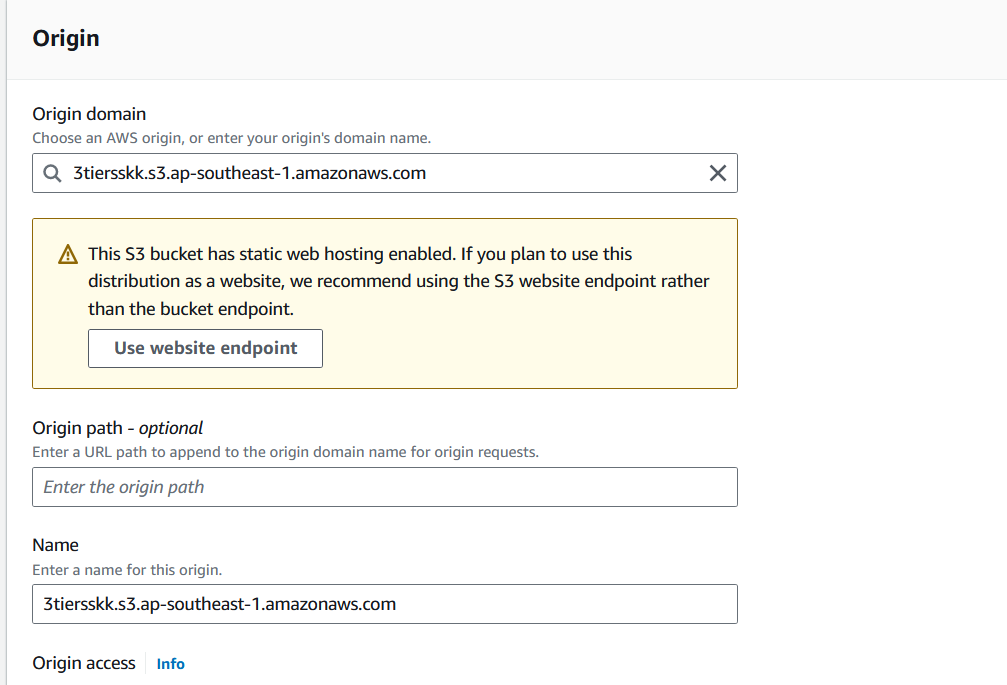




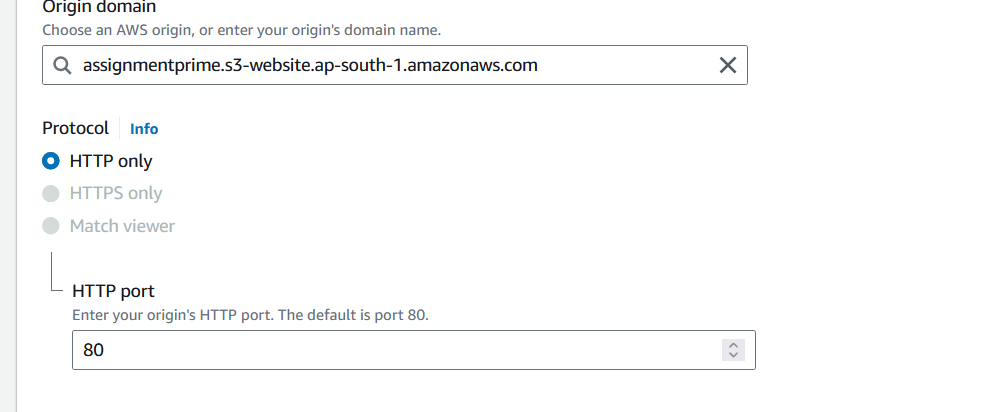




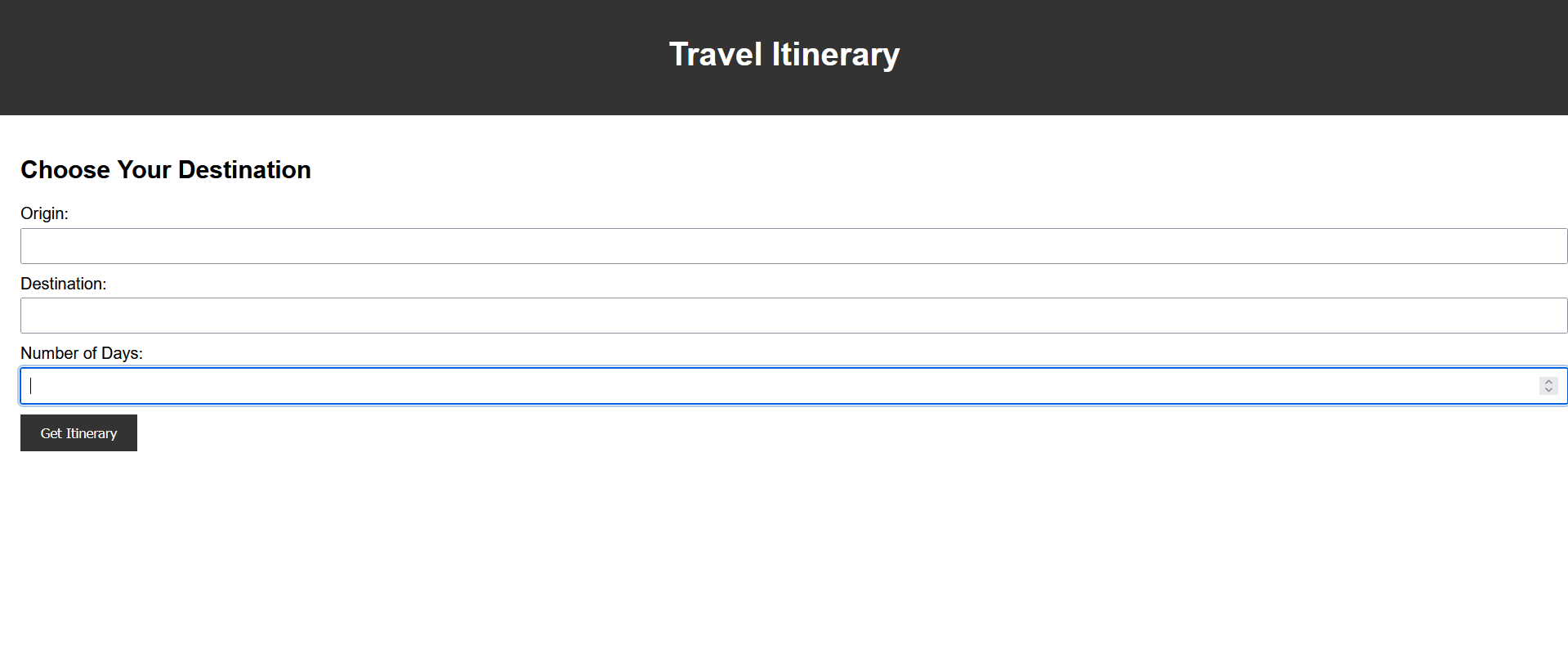
* **Amazon CloudFront**: Use CloudFront as a content delivery network (CDN) to deliver your website content with low latency and high transfer speeds.
* **Create a CloudFront Distribution**: In the AWS Management Console, navigate to the CloudFront service and create a new distribution. Configure the distribution to point to your Amazon S3 bucket containing the website files. CloudFront will serve as a caching layer in front of your S3 bucket.



* **Configure Origin Settings**: Specify your Amazon S3 bucket as the origin for the CloudFront distribution. You can also configure various settings such as caching behavior, origin access identity, and origin custom headers.



* **Set Up Behaviors**: Define behaviors to specify how CloudFront should handle different types of requests (e.g., cache behavior, query string forwarding, etc.). You can customize caching rules and TTL (time-to-live) settings based on your website's requirements.
* **Configure DNS**: Once your CloudFront distribution is created, you'll be provided with a CloudFront domain name (e.g., <distribution-id>.cloudfront.net). Update your DNS settings to point your domain (e.g., www.example.com) to the CloudFront distribution.
* **SSL/TLS Configuration**: Configure SSL/TLS certificates to enable HTTPS support for secure communication between clients and CloudFront.
* **Testing and Monitoring**: Test your website to ensure it's being served through CloudFront. Monitor CloudFront metrics and logs to analyze performance and troubleshoot any issues.

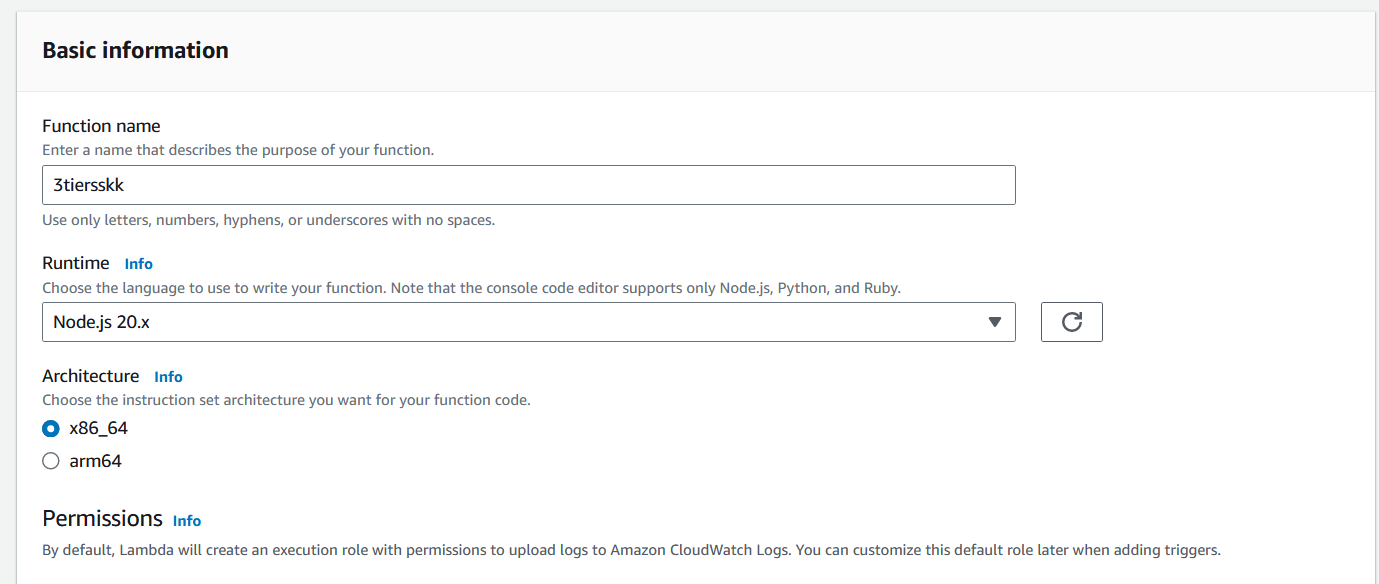


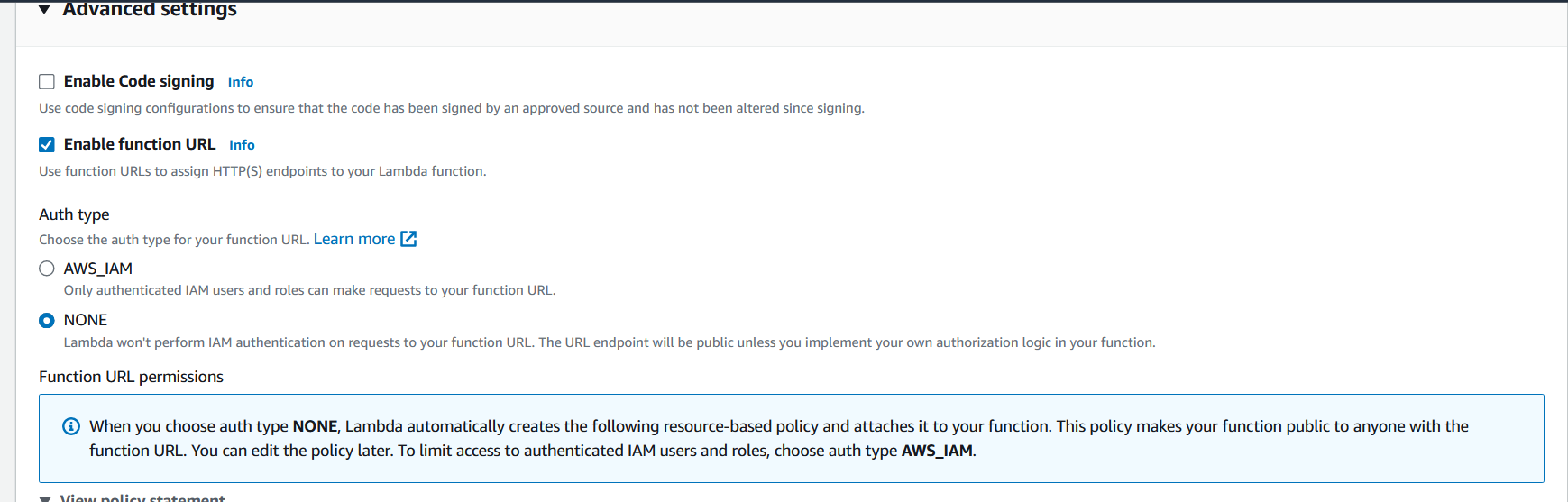
* **AWS Amplify**: You can use AWS Amplify to easily deploy and manage your frontend application. It also provides features like authentication, analytics, and CI/CD.

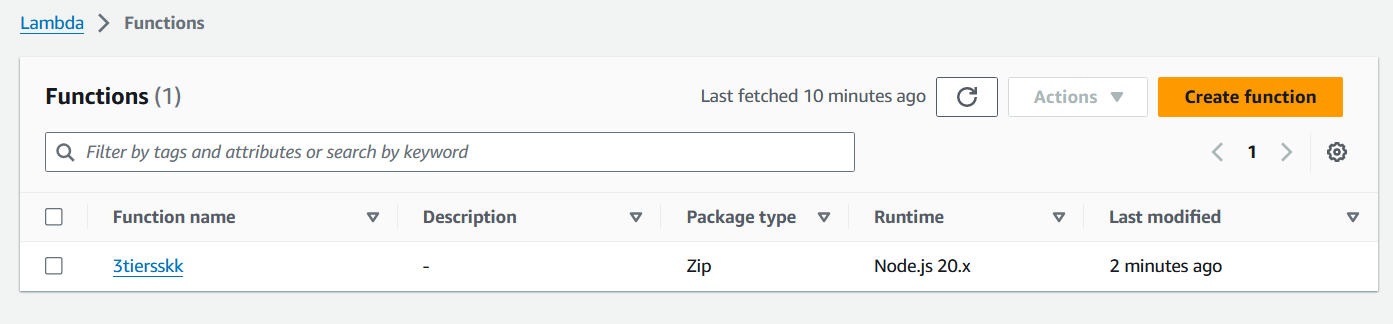
**Backend (Application Tier):**

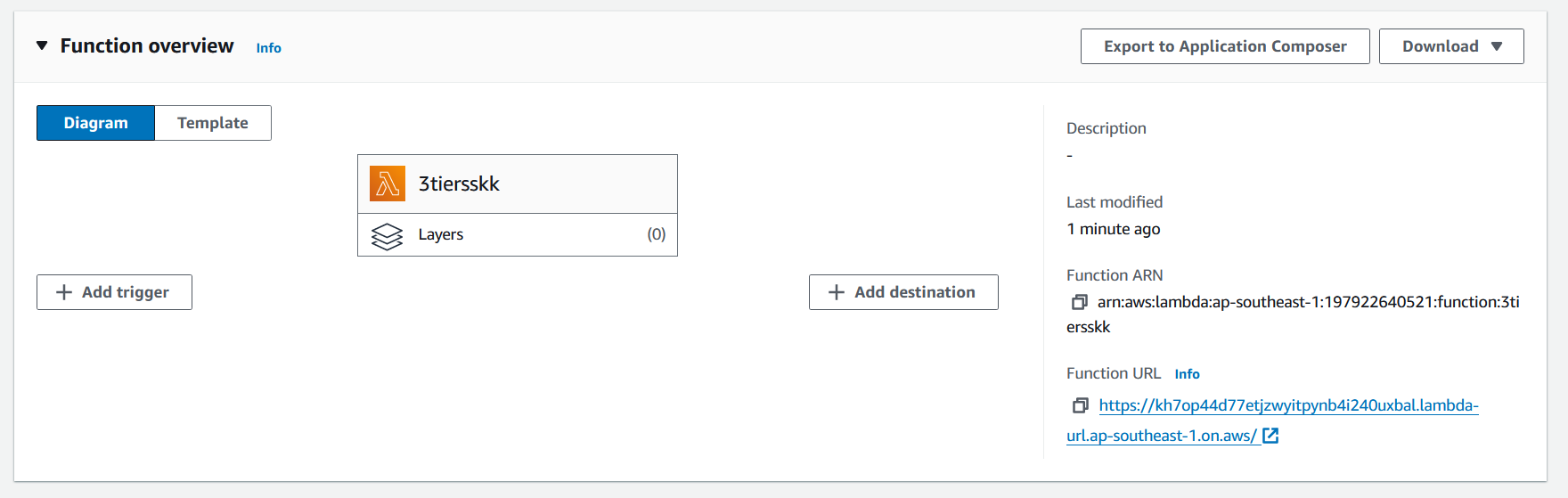
For the backend, you can use AWS services like:

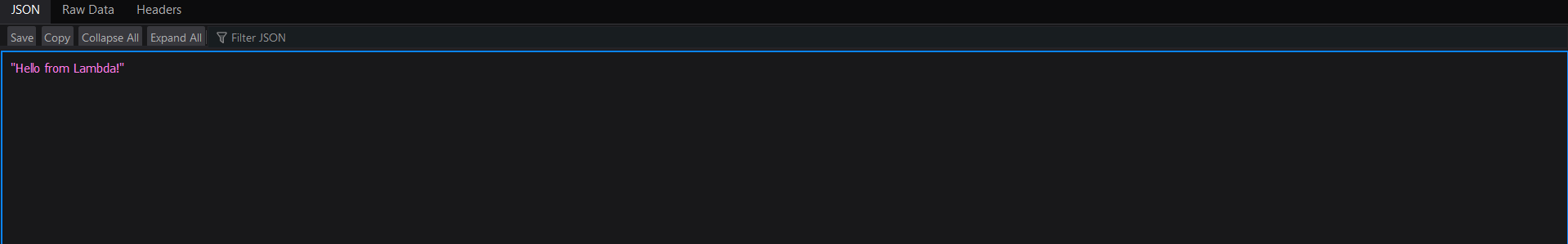
* **AWS Lambda**: Use Lambda functions to handle backend logic, such as processing user requests, interacting with APIs, and generating itineraries.





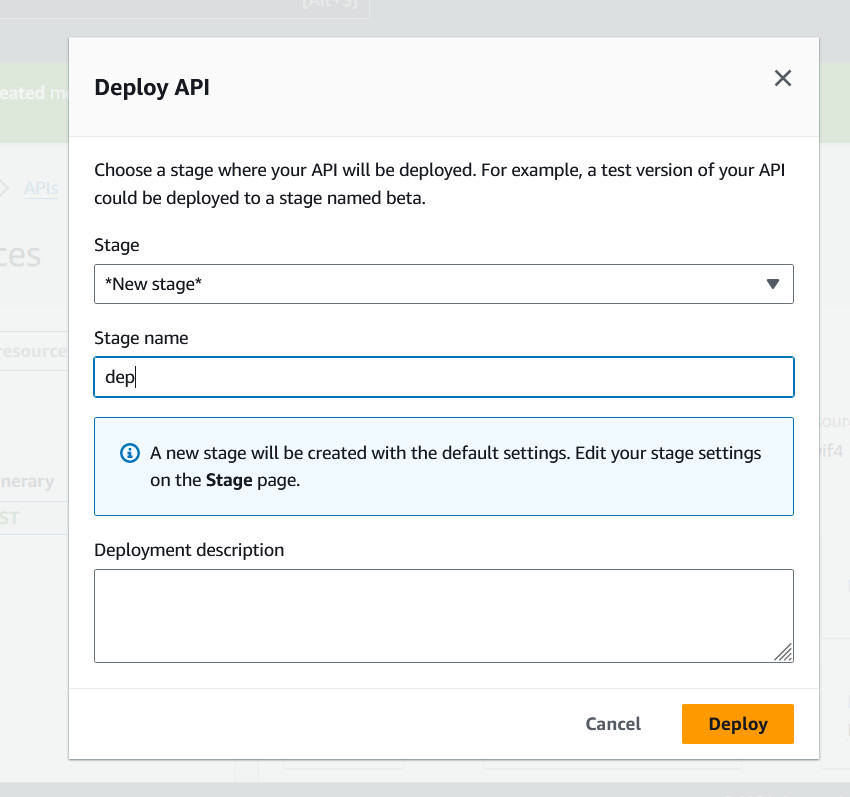
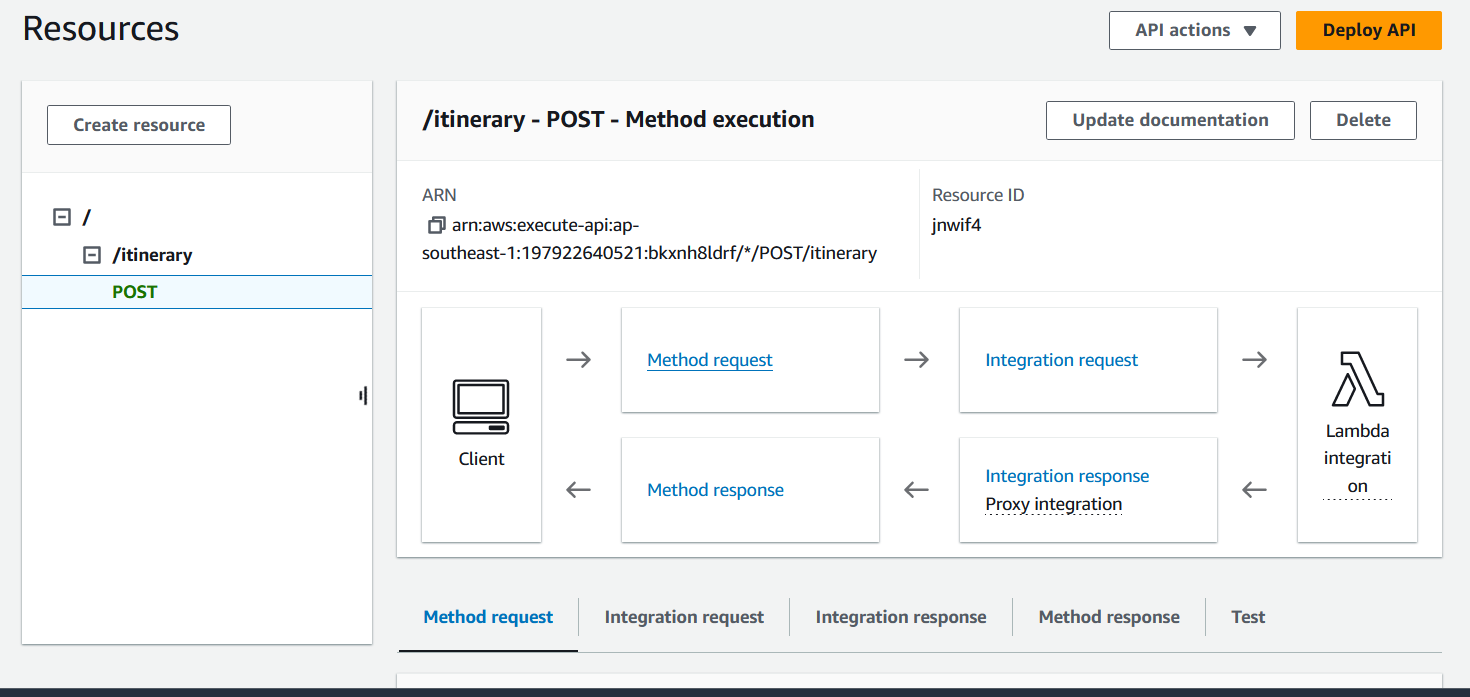
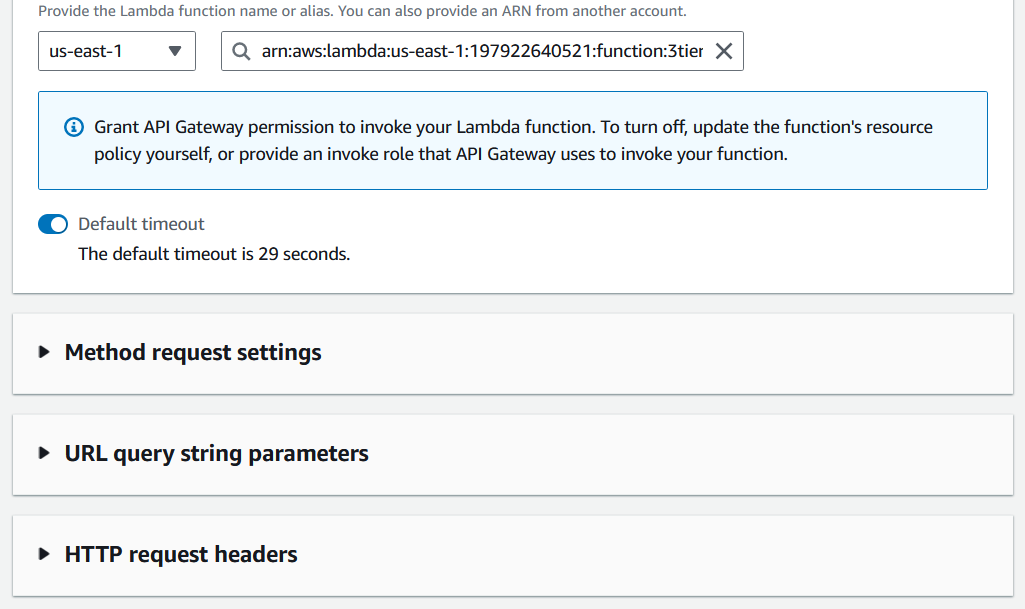
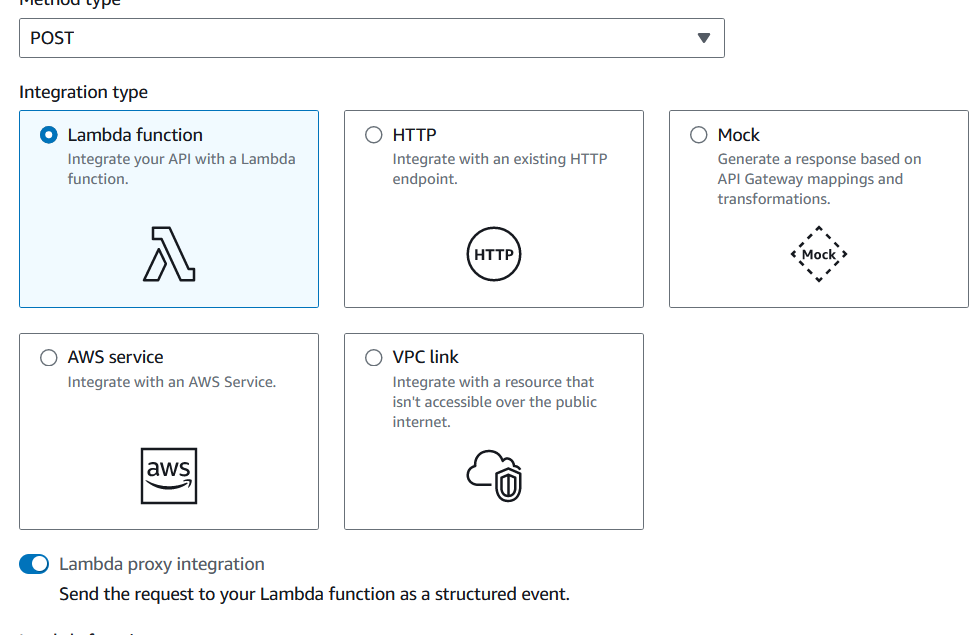
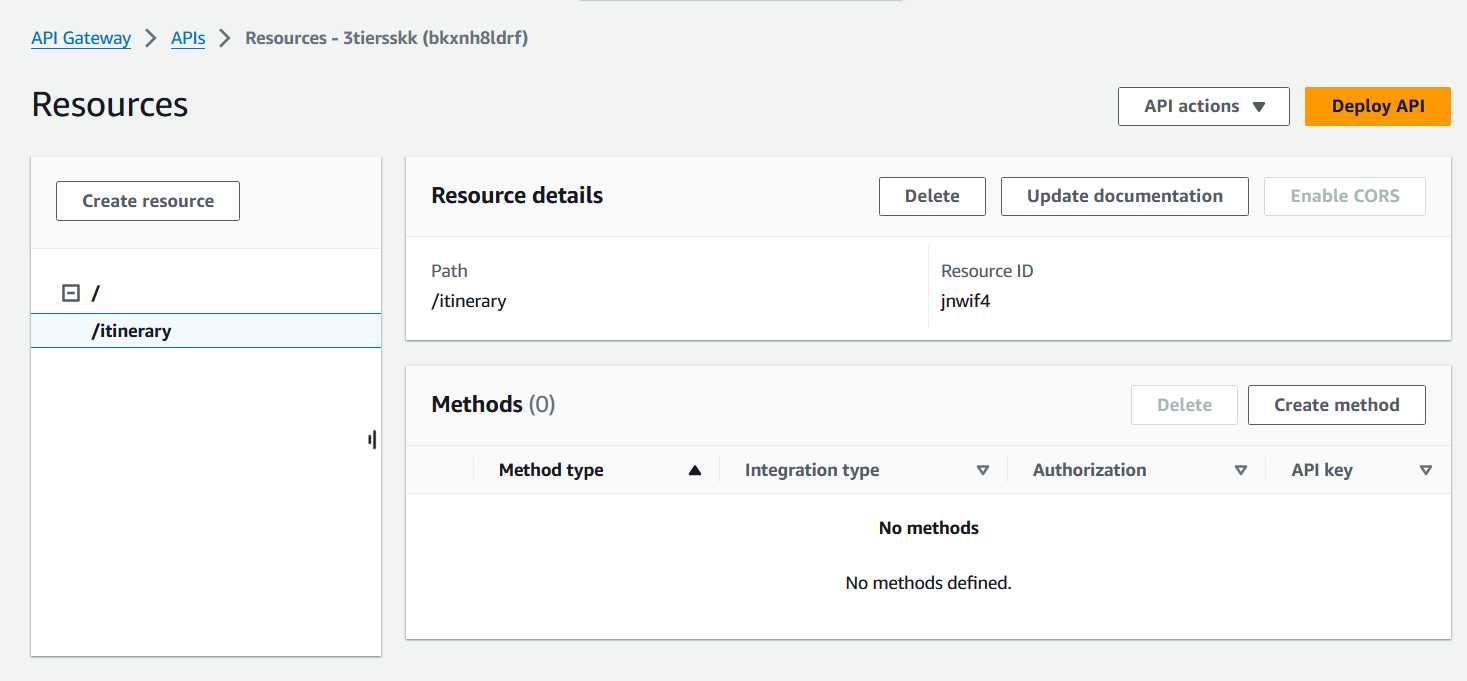
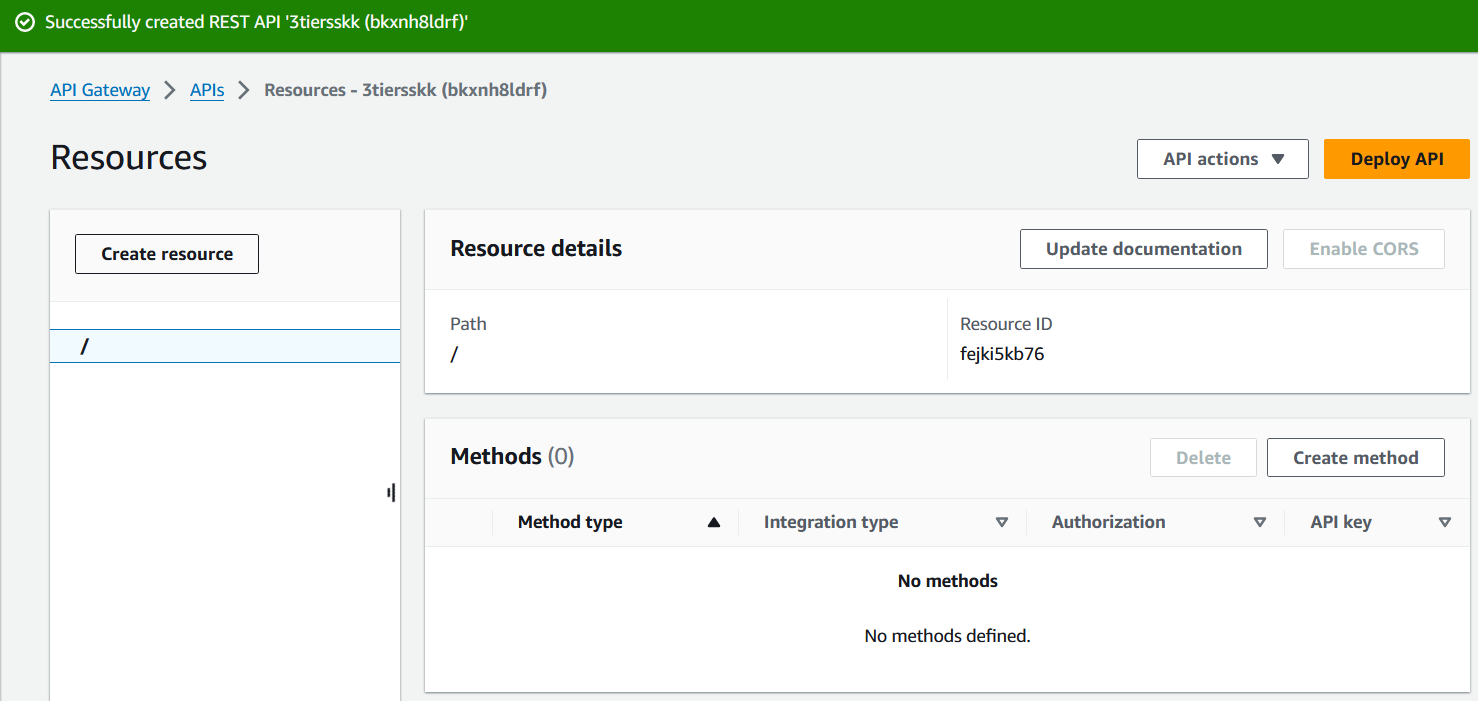
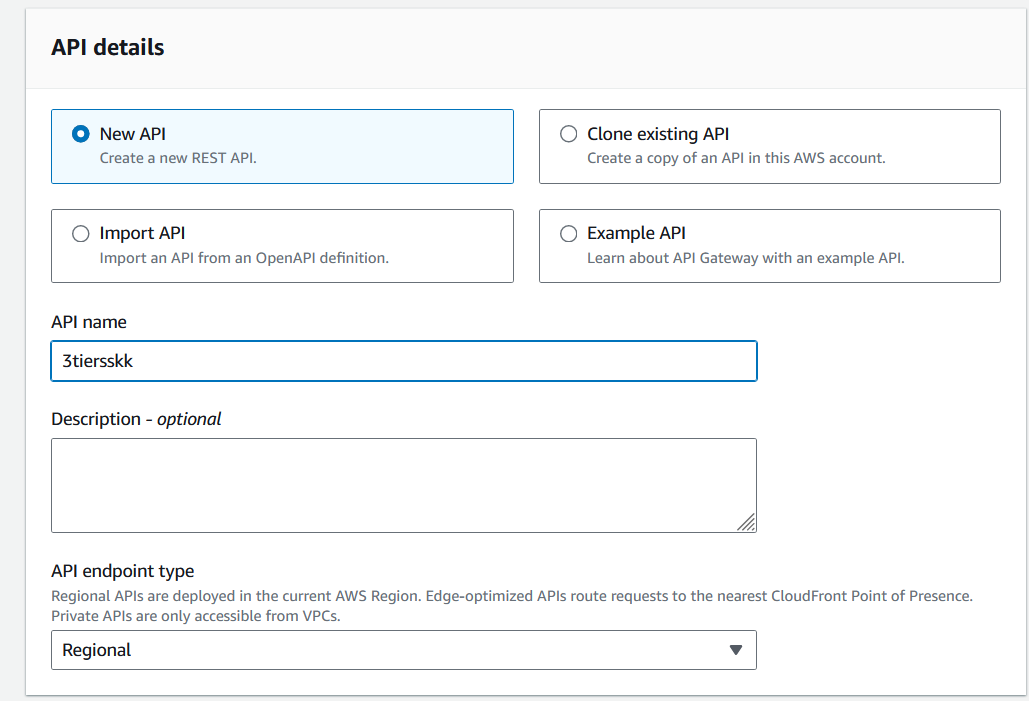
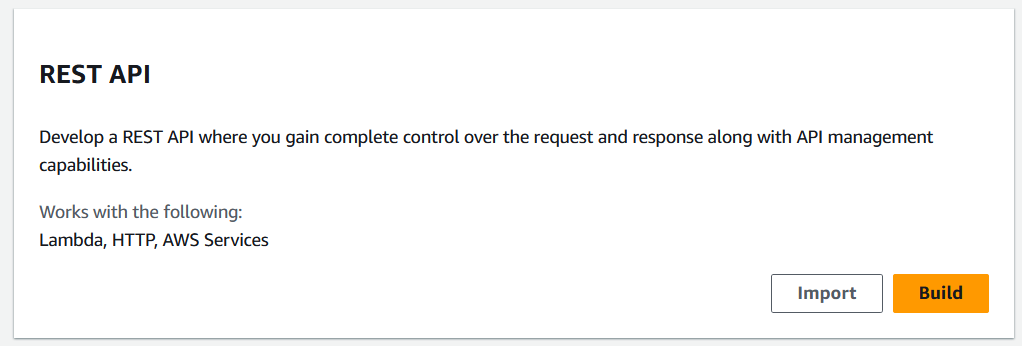
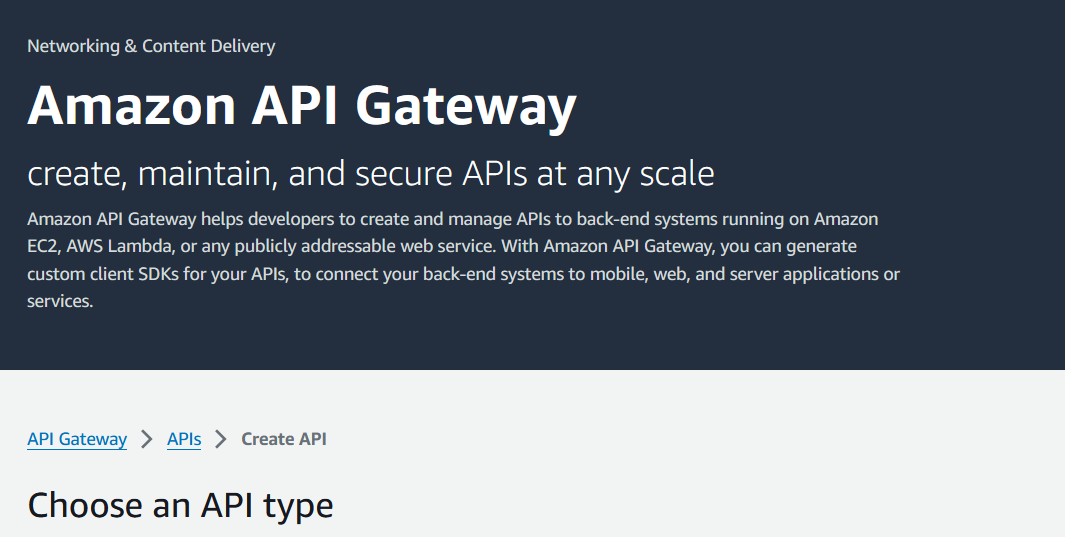






Cloudformation

* **Amazon API Gateway**: Create APIs to communicate between your frontend and backend. API Gateway can trigger Lambda functions based on HTTP requests from the frontend.



* **AWS DynamoDB**: Store data such as user preferences, itineraries, and other application data in DynamoDB, a NoSQL database service provided by AWS.

**Database (Data Tier):**

For the database layer, you can use:

* **Amazon RDS (Relational Database Service)**: If your application requires a relational database, you can use RDS with MySQL, PostgreSQL, or other supported database engines.
* **Amazon DynamoDB**: As mentioned earlier, DynamoDB can be used for storing non-relational data. It's a fully managed NoSQL database service that provides high performance at any scale.

**Architecture Overview:**

1. **User Interaction**: Users interact with the frontend hosted on Amazon S3 through their web browsers.
2. **Frontend-Backend Communication**: Frontend communicates with the backend through API Gateway endpoints.
3. **Backend Processing**: API Gateway triggers Lambda functions that process user requests, fetch data, and generate itineraries.
4. **Data Storage**: Data required for the application is stored in DynamoDB or RDS depending on the requirements.
5. **Scraping External Data**: For scraping itineraries from external sites like Google, you can use libraries like BeautifulSoup or Scrapy within your Lambda functions.

This architecture provides scalability, reliability, and flexibility to your application, and it leverages various AWS services to handle different aspects of your application stack. Make sure to consider security best practices, cost optimization, and performance tuning while designing and deploying your application.