Scenario 1: Logging

I could use a NoSQL database like MongoDB or Elasticsearch to store the log entries. These databases are highly scalable and can handle large amounts of data, making them ideal for logging applications. To allow users to submit log entries, I could create a RESTful API that accepts log entries in JSON format. This API could have authentication and authorization mechanisms to ensure that only authorized users can submit log entries. To allow users to query log entries, I could create a search API that allows users to specify search criteria and returns log entries that match the criteria. This search API could use the indexing capabilities of the NoSQL database to efficiently search through the log entries. To allow users to see their log entries, I could create a user interface that allows users to view their own log entries. This user interface could be a web application built using a front-end framework like React or Angular, and it could consume the RESTful API mentioned above. For the web server, I could use a web framework Express to handle incoming requests and route them to the appropriate API or user interface.

Scenario 2: Expense Reports

To create an expense reporting web application, I would first design the database schema to store the expense data. I would create a table with columns for id, user, isReimbursed, reimbursedBy, submittedOn, paidOn, and amount. The id column would be an auto-incrementing primary key, and the other columns would store the relevant data for each expense. For the web server, I would choose a framework like Django. These frameworks have built-in support for generating PDFs and handling emails, which will make it easier to implement those features. To handle emails, I would use a library like Django's built-in EmailMessage class. When an expense is reimbursed, the web application would generate a PDF of the expense report and attach it to an email, which would then be sent to the user who submitted the expense. To handle PDF generation, I would use a library like ReportLab for Python. These libraries allow for programmatic creation of PDF documents, which will be useful for generating expense reports. For templating the web application, I would use a popular templating language like Django templates. These languages provide a simple syntax for rendering dynamic content in HTML, which will make it easy to display expense data to users

Scenario 3: A Twitter Streaming Safety Service

To build this service, I would use the Twitter streaming API, which allows for real-time access to Twitter data. This API would allow me to monitor tweets in a specified geographic area and scan them for specific keywords. To make the system expandable beyond my local precinct, I would design the application to be modular and scalable. This would involve using containerization technologies such as Docker to package the application and its dependencies into isolated, portable containers. Amazon Web Services (AWS) can also help me host the application and its various components.

To ensure that the system is constantly stable, I would use a combination of automated testing, continuous integration, and monitoring. This would involve setting up automated tests to ensure that new code changes do not introduce bugs, using a continuous integration system such as Jenkins to automatically build and deploy the application, and using a monitoring system such as New Relic or Datadog to monitor the system's performance and detect any issues. For the web server technology, I would use a modern web framework such as Django. This would provide the necessary tools to develop the web-based interfaces for managing triggers, viewing historical data, and accessing the real-time incident report. For the trigger database, I would use a database management system such as MySQL. This would allow me to store trigger combinations in a structured and organized manner, making it easy to query and retrieve information.

For the historical log of tweets, I would use a NoSQL database such as MongoD, which are designed for storing large amounts of unstructured data, making them ideal for storing the vast amount of tweets that would be generated by this system. To handle the real-time, streaming incident report, I would use a combination of websockets and server-sent events. This would allow me to push new tweets to the web-based interface as soon as they are detected, providing real-time updates to users. To handle storing all the media associated with the tweets, I would use cloud-based object storage such as Amazon S3 or Google Cloud Storage. This would allow me to store large amounts of media files in a scalable and cost-effective manner.

Scenario 4: A Mildly Interesting Mobile Application

I would choose MongoDB or Elasticsearch to handle geospatial nature of data. This would allow me to efficiently store and query data based on its location. For storing images, I would use a combination of cloud storage services such as Amazon S3 or Google Cloud Storage for long-term, cheap storage, and a content delivery network (CDN) like Cloudflare or Akamai for short-term, fast retrieval. I would write the API using a language and framework that are well-suited for building APIs, such as Node.js with Express. For the database, I would choose a relational database such as PostgreSQL or MySQL for storing user and administrative data, and a NoSQL database such as MongoDB for storing the 'interesting events' data. This would allow for flexibility in the types of data being stored and efficient querying of large volumes of data.