In order to store the log entries, I would use a NoSQL database like Mongo, since it must support any number of customizable fields for an individual log entry, which does not make sense for a relational database. The schema would have a collection of loggers with common fields, and a subschema for custom subfields for each arbitrary technology. Since the server is expected to be hit by multiple technologies at a time, I would probably use a web framework for the server like Gin for Go that can sufficiently handle the tremendous amounts of requests. For the users of this service, I would expose a REST API as well as a minimal front-end for visualized interaction. The REST endpoints would be on the Gin server so that developers can integrate data flow into their own applications as well as query their logs. As an alternative to the API, the user will be able to query/submit through a front-end implemented in React, served statically through the server. The React interface will allow users to query their logs and submit new entries, . It will also be speedy in the process due to React's reconciliation algorithm and the virtual DOM, which would be necessary for large amounts of entries.

Since the given expenses are always formatted in the same structure, it would be beneficial to use a relational SQL database such as MySQL. Assuming the id attribute of the data is unique, the schema will have that as the primary key, so the data can be properly queried. As for the web server, Node.js with Express would be a good candidate, as there are plentiful templating engines such as Handlebars.js or Pug that will be able to handle all the templating for the web application. As for the pdf generation, Latex would be useful to programmatically template each expense report, as well as look extremely professional. Since sending emails is computationally expensive, it may be better to use a service such as SendGrid to handle actual sending of the emails, which would be controlled by requests from our Express server.

First off, the Twitter Filter Realtime API would be the proper API to use, because it allows to filter Tweets in realtime based on location(https://developer.twitter.com/en/docs/tweets/filter-realtime/guides/basic-stream-parameters). If a tweet is geotagged, the API allows for comma-separated list of longitude, latitude pairs specifying a set of bounding boxes to filter Tweets by. The API delivers the data real-time in a stream which will be collected by a Node.js/Express server. As tweets get sent in, they will be queued and processed via web workers that use low-level C/C++ for parsing to ensure maximum speed. The web workers will analyze the tweets, generate meta-data and send emails (with a service like SendGrid) as well as text messages (with a service like Twilio) to alert based on the contents. In case of disaster-level situations where their is an extreme amount of data being received, the system will be able to spin up more web workers if necessary. Once the web worker is finished analyzing, it will send the tweet data, analyzed metadata alongside with any other binary media data to Apache Cassandra for long-term storage. Cassandra is perfect for logging historical data, because it's a distributed database system that doesn't require an immense amount of management once configured. In addition, if one of the server nodes is having issues, it can be removed and the system will automatically remap the data, therefore ensuring stability and longevity. A front-end client would be implemented with a templating engine for Express such as Pug, which can be used for querying Cassandra using CRUD combinations and displaying the results to the end-user. In addition, the

front end will use sockets to display streaming, online incident reports to the end user. Also, to ensure maximum portability with other precincts and scalability, each component of the system will be put into Docker containers and piloted by Kubernetes. With this system, it will be easy to redeploy per precinct (with simple build scripts for location specifics) and add to the distributed Cassandra cluster.

To handle the geospatial nature of the data, images would be cached for a specific period of data at localized Amazon S3 bucket servers that are closest to the end-user. Choosing AWS regions that are geographically close to the end-user optimize latency, minimize costs, and also address regulatory requirements. In addition, AWS Transfer Acceleration takes advantage of CloudFront's globally distributed edge locations where data is routed to Amazon S3 over an optimized network path. For long-term storage, the images may be backed up to a different data storage provider such as BackBlaze, who offer much cheaper object storage service than AWS S3 but at a slower speed. The API will be using GraphQL, which leverages the best SOAP and REST: strong data enforcement, standardized HTTP to expose data and operations, and the efficiency data payloads. A PostgreSQL database will act as the bridge between user information and their data in their object store. Since we have a basic user to bucket location schema, a relational database like Postgres makes sense. All of this data will be connected with Express/Node.js web servers that are load balanced with NGINX to balance the amount of users each server is handling. The administrative dashboard front-end will be written in React and Relay to easily connect the components to the GraphQL API. Similarly, the client app will be written in React Native and Relay to facilitate the data flow with our GraphQL API.