Scenario 1: Logging

For storing the data, I will a NoSQL database, like MongoDB, as it must support any number of customizable fields for an individual log entry.

For user log entries, I will create an API, which accepts a HTTP POST request containing log data in JSON format.

Users can send HTTP GET requests for querying logs along with the search terms for the log they want to view. We can use Elasticsearch for querying huge log data for quick results.

To view their own log, we can create a HTTP GET API, where it takes user ID from the session and return all the current user log entries. I will create server-side using combination of Express.js and Node.js. For Frontend I will use, React.

For web server, I will use express.js with combination of either Nginx or Apache to increase security. Moreover, they are very compatible with above mentioned tech stack and have good performance. I will use one of these web servers on LINUX virtual machine, most probably, Ubuntu on either Azure or AWS.

Scenario 2: Expense Reports

In this case, as data is structured, I can either use Relational database (like PostgreSQL, or MySQL) or NoSQL (like MongoDB, Amazon DynamoDB, or Cassandra). But to make sure that data is always the same with correct datatype, I will PostgreSQL, with schema that will have following fields: **id, user, isReimbursed, reimbursedBy, submittedOn, paidOn, and amount.**

{

id: ObjectId,

user: String,

isReimbursed: Boolean,

reimbursedBy: String,

submittedOn: String,

paidOn: String,

amount: Number

}

For web server I will use Express.js for its simplicity and robust nature for creating API. I will also use Apache on top of it to increase security layer.

If email is from small to medium number, then I can use Nodemailer, which is highly secure module with OAuth2 authentication. With support of SMTP, it also allows us to use HTML content as well as plain text with embedded image attachments in mail. For bulk mail like, we can also use third party services like SendGrid or Mailgun, as they are powerful APIs that allow us to send, receive, track and store email effortlessly.

I'll be using wkhtmltopdf, a Webkit that prints HTML to PDF, to create pdf documents. Here, we can create a full HTML document with images, CSS, and other features, and submit it to wkhtmltopdf to be converted into a PDF.

Lastly, I'll be using handlebars for templating because it enables us to keep your HTML pages simple, clean, and independent of the logic-based JS files.

Scenario 3: A Twitter Streaming Safety Service

I plan to utilize the standard Twitter API in combination with Node.js to make requests to the Twitter API.

To expand beyond a localized area, I will implement MongoDB which enables us to adapt the database to meet our specific requirements. Moreover, I would deploy it on a cloud platform such as AWS and use load-balancer for handling huge traffic flow.

To ensure system stability, I would implement monitoring and alerting using a tool such as Nagios and use automated testing scripts written in selenium (python) for rigorous testing.

Next.js will be used for web server because it is light in weight, has great adaptability and responsiveness, security, shorter load time, improved SEO, a faster development process, and better image optimisation.

For triggers, I will use Elasticsearch that stores data in indexes and supports powerful searching capabilities. To keep a record of past tweets, I will use MongoDB because it is versatile and permits data storage in unstructured JSON format.

React.js will be used for streaming incident reports because it is making it possible to update specific components in real-time due to its component-based nature, and it also supports Next.js.

Amazon S3 bucket will be used for persistent file storage, with CloudFront as CDN. EmailJS will be used to send emails as it is supported by React. Moreover, it is a single module with zero dependencies – code is easily auditable. The email delivery use TLS/STARTTLS for secure email transmission.

Scenario 4: A Mildly Interesting Mobile Application

To handle the geospatial nature of data, I save location information available in image metadata using database like MongoDB which support Geospatial queries and indexing. My data will look like this:

```
{
        “_id”: “63a5d9785c3c5f0149eec9c6”,
        “image”: “location.jpeg”,
        “location”: {
                “type”: "Point",
                “coordinates”: [-73.856077, 40.848447]
        }
}
```

For storage, I will use Amazon S3 buckets, as they are cost-effective and provide high durability for long term. On top of it, I will content delivery network (CDN) such as Amazon CloudFront, which can cache content closer to the user, reducing longer loading time. I will also use ImageMagick with node.js to compress and resize image when user uploads it, so that small size of image can also contribute of faster retrieval of the file.

I will write my API in Node.js and use express.js to make API calls. I will take user location from there device and make an API call querying the mildly interesting photos of that location and return it to user.

I would combine a frontend framework like React with a UI library like Bootstrap or Material Design to create the management dashboard.