Scenario 1: Logging

To implement this scenario, I would be using Express in Node.js as the web server and Winston, Morgan for logging and MongoDB to store the logs. I have chosen the above because of the following reasons:

Express: Although Node offers built-in modules to create HTTP server and routes, Express makes it much simpler to perform the same functionality. In addition, it provides a simple syntax to implement multiple routes, and offers various built-in and third-party middleware. Using these middleware, the app functionality and support can be expanded to accommodate cookies, sessions and even logs.

Morgan: Morgan is an HTTP request middleware logger for Node.js that automatically logs incoming requests to the server, such as the request method, response status, user IP address, etc. and generates logs in the specified format. Using Morgan eliminates the need to create a custom middleware for the same purpose.

Winston: Winston is a very popular Node module used for logging purposes. It is relatively stable compared to other libraries and offers a lot of customizability through its configuration options, thus making development seamless. Winston also offers different options to export the log data, either to a file, console, or a third-party application.

MongoDB: Through the Winston-mongodb npm module, the application can export the logs to a MongoDB collection as json data. Users can filter records matching criteria using mongodb's filtering methods and can even export them, if necessary.

Scenario 2: Expense Reports

To implement this scenario and fulfil all the requirements, I would use Express as the web server, PostgreSQL to store the expense report details, and the node modules *pdfkit* and *nodemailer* to generate the PDF and send the email, respectively. For the HTML content, I would create Handlebar templates for user login, submission and tracking of expenses and one for the payroll team to reimburse the expense.

Express: Express provides a simpler syntax to implement multiple routes and offers various built-in and third-party middleware. Using these middleware, the app functionality and support can be expanded to accommodate cookies, sessions and even logs.

PostgreSQL: As the fields do not change, I have opted to use PostgreSQL as the storage medium. Using the *knex* npm module, node applications can perform CRUD operations and data can be returned in a json format.

Pdfkit: pdfkit makes adding text to documents quite simple, and includes many options to customize the display of the output. Adding text to a document is as simple as calling the text method. If no options are given, pdfkit automatically wraps text within the page margins and places it in the document flow below any previous text, or at the top of the page.

Nodemailer: Nodemailer can be used to create HTML or plain-text emails, add attachments (both static and dynamic), and send your emails through different transport methods, including built-in SMTP support. For development purposes, a fake SMTP server can be created using mailtrap

Scenario 3: A Twitter Streaming Safety Service

I would build the user interface using ReactJS as it has the capability to efficiently update and render only those components where the data has been updated.

As for the scalability of the application i.e., to expand beyond the local precinct and to stabilize the application, both features can be implemented using the Amazon Elastic Container Service (ECS) product offered by AWS. With the help of Amazon CloudWatch, the users can monitor the CPU and memory utilized by the ECS resources to determine if additional instances to handle increased load or even stop under-used instances to save money.

To filter tweets returning only those that match the trigger, the rules need to be created first, which is done by the API:

POST /2/tweets/search/stream/rules

Where the trigger keywords are added as part of the request body. Once the rules are added, the endpoint

GET /tweets/search/stream

is run to retrieve the filtered tweets in real-time through a persistent streaming connection. The stream does not need to be disconnected to add or remove rules.

To handle the long-term storage of media and readily view the data, Amazon DynamoDB can be implemented that would satisfy the storage constraint. It also offers in-memory caching and data export functionality through which users can store data and media for longer periods of time as well as retrieve and view historical events and reports when needed.

The underlying web server technology that connects the UI to the cloud would be Node.JS and Express as they have a vibrant and sophisticated community with over 1 million npm modules. We can use ***aws-sdk*** to connect to the AWS platform and ***react*** and ***react-dom*** to connect to the UI. To send text alerts to different officers based on the triggers, the Twilio Messaging API can be used through the ***twilio*** module to send messages seamlessly. Similarly, for the email alerting system, ***nodemailer*** module can be used to send emails to the concerned officers depending on the contents of the tweet and who tweeted it.

Scenario 4: A Mildly Interesting Mobile Application

The back-end of the application can be written in NodeJS and Express where we can use the npm modules ***multer*** and ***multer-gridfs-storage*** to convert images into multipart and store them as files as part of an object.

Using the built-in Geolocation API, we can obtain the location of the user, which can be stored along with the image as part of the same object. The administrative dashboard to manage the content can be built using React where we can update the component to display images of interesting events.

Short term storage and fast retrieval functionality can be implemented using Redis cache which also supports geospatial indexing by encoding the latitude and longitude coordinates into the score of the sorted set using the geohash algorithm. For cheaper storage for longer periods of time, a NoSQL database such as MongoDB can be used for file storage, where the images can be stored and retrieved whenever necessary using the GridFS storage module.