Scenario 1: Logging

Store log entries: NoSQL databases like MongoDB or Elasticsearch which allows flexible schema design and supports JSON like documents, ability to scale horizontally, and support for indexing and searching

Allow users to submit log entries: RESTful API using web framework like Flask which can handle HTTP request and responses

Query log entries: Elasticsearch which is designed for full-text search and analysis, users can see the logged entries by querying the database through the API. Elasticsearch is a scalable, open-source search and analytics engine. It supports real-time indexing and searching of log entries and is simple to combine with other technologies.

Web search: NGINX which is a lightweight and efficient web server that can also function as a reverse proxy for the flask application.

The above-mentioned setup can handle high levels of traffic and is easily scalable.

Scenario 2: Expense Reports

Store expenses: relational database like PostgreSQL or MySQL which can handle complex queries and has good support for transactions. Each expense would be represented as a row in a table with the specific fields (id, user, isReimbursed, reimbursedBy, submittedOn, paidOn, and amount) as columns.

Web Server: Django which is a full-stack web framework that includes an ORM for interfacing with the database and a templating engine for rendering HTML templates

Handling Emails: third-party service like SendGrid or MailGun which provide APIs for sending transactional emails. These libraries provide a simple way to send emails through a server, including generating PDFs and attaching them to the email.

PDF Generation: library like ReposatLab or PyPDF which can generate PDFs from python code

Templating: Django's build-in templating engine which supports template inheritance and variable substitution. This would allow us to create a user-friendly web interface for submitting expenses, viewing expense reports, and managing user accounts.

Scenario 3: A Twitter Streaming Safety Service

Access Twitter API: Twitter API library for python which provides a convenient interface for interacting with the API and a message broker such as RabbitMQ to handle the high volume of incoming data

Make the system expandable beyond the local precinct: Cloud-based infrastructure like AWS or Google Cloud that can handle high levels of traffic and provide scalable storage and computing resources. Also, microservices architecture with Docker containers orchestrated by Kubernetes to make the system scalable and maintainable

Web server: Node.js and Express.js for its event-driven, non-blocking I/O model, which is well-suited for handling large numbers of concurrent connections

Databases: Elasticsearch for triggers and MongoDB for historical log of tweets which supports flexible scheme design and fast query times.

Handle real time streaming incident report: WebSockets which enable bidirectional communication between the server and client.

Handle storing media: cloud-based object storage service like Amazon S3 or Google Cloud Storage

Scenario 4: A Mildly Interesting Mobile Application

Handle geospatial nature of data: Database with geospatial indexing support like PostgreSQL with PostGIS or MongoDB with GeoJSON

Long-term cheap storage of images: Cloud-based object storage service like Amazon S3 of Google Cloud Storage

Short-term fast retrieval of images: Content Delivery Network (CDN) like Cloudflare or Akamai which can cache images and serve them from the nearest edge location

API: Web framework like Node.js and Express.js for its scalability and ease of use

Database: PostgreSQL or MongoDB depending on the data structure and complexity of queries

Administrative dashboard: front end framework like React or Angular which can interface with the API and display data in a user-friendly way