**Question 1.** How would you scale up the solution to a large number of devices uploading data and higher throughput? What problems do you expect to encounter and how would you solve them? Describe them in the context of your solution?

To scale up the current solution for a large number of devices uploading data at higher throughput, several challenges would need to be addressed, including performance, memory management, concurrency, and fault tolerance. Here's a breakdown of the key problems and their corresponding solutions:

**Improvements**

1. **Concurrency:** Use asynchronous/multithreaded processing and message queues to handle high throughput (e.g., Kafka).
2. **Data Storage:** Leverage distributed in-memory caching and databases to store profiles and state. Store device profiles and states in a distributed in-memory cache system such as Redis. Periodically persist the event counts and device states to a database to avoid data loss in case of failure.
3. **Scaling:** Use load balancing, microservices, and horizontal scaling to distribute processing.
4. **Fault Tolerance:** Implement checkpointing, redundancy, and idempotency to handle system failures.
5. **Streaming:** Integrate with real-time data streaming platforms for efficient large-scale data processing. Apache Flink for distributed, real-time data processing.
6. **Monitoring:** Use robust monitoring tools to track system health and performance. Implement monitoring tools such as Prometheus, Grafana (e.g., event processing latency, memory usage, throughput). Set up alerting mechanisms for threshold violations (e.g., too many devices missing profile data or excessive active windows
7. **Optimization:** Improve event aggregation and reporting with scalable database solutions.

**Question 2.** How would you implement the persistence layer of your solution? Describe the technologies in detail and provide reasoning for your selection.

1. **PostgreSQL (with sharding/partitioning):** Structured storage of device profiles and metadata.
2. **Redis (as a cache):** Fast in-memory lookup for frequently accessed device profiles and state.
3. **Cassandra/ScyllaDB:** High-throughput, distributed storage for event data and time-series information.
4. **InfluxDB:** Optimized time-series database for real-time event logs.
5. **Apache Kafka:** Message broker for data ingestion and decoupling producers and consumers.
6. **Amazon S3:** Low-cost object storage for long-term data archiving.
7. **Prometheus + Grafana:** Real-time monitoring and alerting for system health.
8. **Backup Strategies (Snapshots, Replication):** Ensuring disaster recovery and data durability.

**Question 3.** How would you make the scaled-up solution fault tolerance and how would you deploy and maintain the solution? Describe the used technologies and their roles. What problems do you expect to encounter and how would you solve them? What could be the alternative solutions?

**Technology:** **Kubernetes (K8s)** or **Docker Swarm** for container orchestration.

* **Role:**

1. **Pod replication:** By deploying the DeviceEventProcessor and other services in containers managed by Kubernetes, multiple instances (pods) can run concurrently across different nodes. Kubernetes ensures that if one instance fails, another is available to handle requests.
2. **Auto-scaling:** Kubernetes can automatically scale the number of pods up or down based on the system's load (e.g., the number of devices sending data). If the traffic increases suddenly, more instances can be spun up to handle the load.
3. **Self-healing:** Kubernetes has self-healing capabilities. If a pod or container crashes, Kubernetes will automatically restart it and ensure the desired number of replicas is maintained.
4. **Rolling updates:** To deploy updates without downtime, Kubernetes supports rolling updates. This ensures that new versions of your application are gradually deployed, replacing old ones while keeping the system functional.

* **Problem to Expect:** If not configured properly, you might encounter **network partitioning** (split-brain scenarios where nodes can’t communicate). Solving this requires configuring **quorum-based** consensus mechanisms (e.g., Raft in Kubernetes) and setting appropriate thresholds for auto-scaling.

**Solution:** Use Kubernetes’ built-in service discovery and load balancing mechanisms to ensure nodes are correctly communicating. For deployment, utilize **blue-green** or **canary deployments** for safe updates with rollback capabilities in case of failures.

**Alternative:** Use **AWS Elastic Container Service (ECS)** or **Google Kubernetes Engine (GKE)** for container orchestration without managing the underlying infrastructure, offloading some of the operational overhead to the cloud provider

**Deployment and Maintenance Strategy**: CI/CD Pipelines (Jenkins, GitLab CI, AWS CodePipeline).

**Question 4.** How would you monitor the health and performance of your system in operation? What metrics would you choose and why? What concrete tools and frameworks would you use?

* **Prometheus** for real-time metrics collection and alerting. Prometheus is a highly popular open-source monitoring tool designed for real-time metrics collection.
* **Grafana** for dashboard visualizations. Grafana integrates seamlessly with Prometheus and provides a rich, customizable dashboard for visualizing the system’s health and performance. It supports real-time visualizations, allowing teams to monitor key metrics at a glance.
* **Alertmanager** for managing alerts and notifications. Prometheus integrates with **Alertmanager** to handle alerts. It supports routing, deduplication, and silencing, making it a robust solution for managing system alerts.
* **Elastic Stack (ELK)** for centralized logging and log analysis. **(ELK: Elasticsearch, Logstash, and Kibana)** is widely used for centralized logging. It allows for real-time log analysis, which is crucial for debugging and understanding system issues.