Question 1

- Why does Prometheus use a pull-based approach for metric collection? And can you think of scenarios where this might not work well?
- Why Pull?
    - Decoupling of the monitoring system from targets.
    - Dynamic discovery of targets in ever-changing environments.
    - Simplified configuration since targets only need to expose an endpoint.
    - Efficient metric selection (enabling the server to choose which metrics to collect).
    - Better scalability for large systems.
- Where It May Fall Short:
    - Short-lived jobs—where the job might complete before Prometheus gets a chance to scrape (solution: use the Pushgateway).
    - Firewalled or restricted networks where a pull isn't possible (solution: proxy access or even push-based alternatives).
    - Highly distributed or IoT systems with unreliable network connections.
    - Event-driven metrics where data is not continuously available.

Question 2

- What happens to your monitoring system if a Prometheus server goes down? How would you ensure its reliability?
- Immediate Effects:
    - Loss of metric collection.
    - Alerts stop firing.
    - Dashboards and query capabilities go offline.
- Reliability Enhancements:
    - Setting up high availability (HA) by running multiple Prometheus instances in parallel—possibly behind a load balancer or ALB for read requests.
    - Using remote write/read capabilities with long-term storage solutions such as Thanos or VictoriaMetrics.
    - Performing frequent backups.
    - Running Alertmanager in HA mode.
    - In Kubernetes, leveraging an operator for automated recovery.
    - Splitting workloads for larger environments.

Question 3

- How would you decide the `scrape_interval` for different types of applications (e.g., a web server vs. a batch job)?
- Factors to Consider:
    i. Application Behavior & Metrics Volatility:
        - High-traffic web servers: metrics change frequently (scrape interval of 5–15 seconds).
        - Batch jobs: scheduled, less frequent execution (intervals of 60 seconds or more).
    ii. Criticality of Metrics:
        - More critical applications might need shorter intervals.
    iii. Metric Retention Requirements:
        - Shorter intervals yield more data, increasing storage needs.
    iv. Resource Constraints:
        - Both on Prometheus and on the target system—the load from frequent scrapes must be balanced.
    v. Types of Metrics:
        - For counters, gauges, and histograms there can be different optimal intervals.

Question 4

- What is the difference between using static_configs and service discovery in Prometheus scraping?
- Static_configs:
    - Targets are manually specified.
    - Works well for a small or fixed set of endpoints.
    - Simple and independent from external systems—but it can be manual and error prone.
- Service Discovery:
    - Automatically discovers targets in dynamic, constantly changing environments (e.g., Kubernetes, AWS EC2).
    - Provides dynamic updates and can also enrich metrics with labels.
    - Introduces dependencies on external systems and increased configuration complexity.

Question 5

- If you see gaps in your metrics data, what are some possible reasons related to Prometheus' scraping process?
- Potential Causes:

i. Target Issues:
    - The target is unreachable, unresponsive, or undergoing restarts.
    - Firewall rules or DNS issues may block access.
ii. Scrape Timeout:
    - The target might be slow or overloaded.
iii. Resource Constraints on Prometheus:
    - High CPU, memory, or disk I/O usage may cause missed scrapes.
iv. Configuration Problems:
    - Overlapping scrape intervals and evaluation intervals.
v. Misconfigured Retention Policies:
    - Causing older data to be dropped.
vi. Exporter or Library Issues:
    - Bugs or misconfigurations in the exporter.
vii. Time Synchronization:
    - Incorrect system clocks can lead to stale or inconsistent data.

Question 6

- Can you think of a scenario where over-scraping might negatively impact your system performance?

1. Impacts on Targets:
    - High CPU and memory usage on the target systems.
2. Network Overload:
    - Excessive traffic may cause network congestion, especially in remote or distributed environments.
3. Strain on the Prometheus Server:
    - Handling too many scrapes may overload the Prometheus server itself.
4. Redundant Data Collection:
    - Gathering data more frequently than necessary may generate redundant metrics, increasing query complexity and storage costs.

- Example:
    - Scraping 500 remote instances every second could overwhelm inter-region network links, or monitoring 10,000 microservices at high frequency can strain the central server.

Question 7

- How would you handle label conflicts when aggregating metrics from different services?
- Approaches:

1. Normalize label names across services (e.g., standardize naming from “host” and “instance” to a single convention).
2. Use Prometheus’ label relabeling rules at scrape time to rename, replace, or drop conflicting labels.
3. In PromQL queries, use functions like label_replace() to handle discrepancies.
4. Introduce namespaces or additional labels to differentiate the source of metrics.
5. Maintain and share documentation for metric conventions with your team.

Question 8

- What are the limitations of Prometheus in monitoring large-scale systems, and how would you address them?
- Limitations:
    i. Scalability challenges as Prometheus is inherently single-node.
    ii. Lack of long-term storage.
    iii. Handling high cardinality metrics.
    iv. No built-in authentication and authorization.
    v. Performance issues with complex, large-scale queries.
    vi. Challenges in multi-region monitoring and potential alert fatigue.
- Potential Solutions:
    i. Use Prometheus federation, or deploy Thanos/Cortex for horizontal scaling.
    ii. Integrate remote storage systems via remote_write/remote_read.
    iii. Limit high-cardinality labels; use aggregation.
    iv. Secure endpoints externally via reverse proxies.
    v. Implement recording rules and optimize PromQL queries.

Question 9

- If a service is exposing metrics in JSON, how would you integrate it with Prometheus?
- Answer
    - Use a JSON exporter that converts JSON output into the Prometheus exposition format.
    - Develop a custom exporter if one isn’t available.
    - Preprocess JSON data with ETL tools (like Fluentd or Logstash) to transform data before exposing it.
    - As an alternative, modify the service itself to support the Prometheus format if possible.

Question 10

- Can you think of a scenario where exposing metrics in the Prometheus format might not be feasible?
- Scenarios:
    i. In high-security or restricted environments where outbound HTTP access is blocked.
    ii. When dealing with an extremely large volume of metrics that could overwhelm Prometheus.
    iii. For dynamic or ephemeral workloads where instances appear and disappear too quickly.
    iv. On resource-constrained devices (e.g., IoT or embedded systems) where running an HTTP server isn't practical.
    v. When an application uses non-HTTP protocols (such as MQTT or gRPC) or is a legacy system.
- Possible solutions:
    i. Use the Pushgateway to push metrics rather than scrape them.
    ii. Employ ETL pipelines to transform data.
    iii. Use sidecar containers or custom scripts for legacy environments.

Question 11

- What are the challenges of securing Prometheus in a cloud-based multi-tenant environment?
- Challenges:
    i. Data isolation between tenants.
    ii. Ensuring proper authentication and authorization.
    iii. Securing data in transit (e.g., with TLS/SSL) and at rest.
    iv. Managing multi-tenant query access.
    v. Securing the exporters that feed data to Prometheus.
    vi. Dealing with scalability and resource contention while meeting regulatory compliance.

Question 12

- How does rate() differ from sum() when aggregating metrics?

1. rate()
    - Computes the per-second average rate of increase for a counter over a time interval.
    - Shows trends in how quickly a counter is increasing.

2. sum()
    - Totals up the values across one or more dimensions.
    - Useful for obtaining an overall count or combined metric value.

Question 13

- When would you use avg_over_time() versus max_over_time()? Can you give examples?

1. avg_over_time()
    - Calculates the average value of a metric over a specified time range.
    - Useful for smoothing out fluctuations—ideal for understanding consistent load.
2. max_over_time()
    - Extracts the maximum value over a time period.

Question 14

- Why is rate() essential for monitoring counters? Can you think of a scenario where it wouldn't give accurate results?
- answer:
    - Counters only increase (or reset), so knowing the rate of increase is key to understanding how quickly an event is occurring.
    - Without rate(), raw counter values might be misleading, especially after a reset—they could appear negative or unusually low.
    - A scenario where rate() might not be accurate is when a counter resets unexpectedly; proper handling (such as using the irate() function for short-term analysis) might be needed to catch that behavior.

Question 15

- What's the difference between irate() and rate()? Which one would you use for short-term analysis?
- answer:
    - rate()
        - Averages the per-second increase over an entire time range, smoothing out fluctuations.
    - irate()
        - Calculates the per-second rate of increase for a counter over a time interval.

        - More responsive to immediate changes and spikes; ideal for short-term analysis.

Question 16

- When would you choose a histogram over a gauge for metrics collection?
- answer:
    - Histograms are ideal when you want to capture the distribution of values across multiple buckets (for example, HTTP request durations).
    - They allow you to compute percentiles or quantiles and analyze the spread of data.
    - Gauges, in contrast, are used for instantaneous measurements (like current memory usage).

Question 17

- What is a Prometheus histogram, and how would you interpret its bucket metrics?
- answer:
    - A histogram is a metric type that divides observed values into buckets, each representing a range (e.g., request durations ≤0.1s, ≤0.2s, etc.).
    - Each bucket counts the number of events falling within its range.
    - There is also a cumulative count (with the le label) that shows “less than or equal” to a threshold.

Question 18

- How would you troubleshoot a metric that appears stale based on its timestamp?
- Answer:
    i. Check the scrape interval and job configuration.
    ii. Verify that the target is available and reachable (watch for restarts or network issues).
    iii. Review Prometheus logs for any scrape errors or timeouts.
    iv. Validate time synchronization (using NTP, for example) between the server and targets.
    v. Inspect the metric endpoint for missing or outdated data.
    vi. Consider misconfigurations in retention policies or issues with label consistency.

Question 19

- How does Prometheus handle timestamps, and why are they important in queries?
- How:
    - Prometheus assigns a timestamp to every data point when it is scraped, and these are stored in TSDB.
    - Timestamps are used to correlate data points and ensure that calculations (like rate or averaging over time) consider the proper time windows.
- Why:
    - They are essential for precise time-based analyses, ensuring that queries and functions compute data over the intended intervals.
    - Timestamps are key to handling missing or stale data, and they enable functions like rate(), avg_over_time(), etc., to work correctly.

Question 20

- How would you approach optimizing a Prometheus setup for a high-traffic production environment?
- Answer:
    i. Scaling and Sharding:
        - Use Prometheus federation or horizontal scaling techniques (e.g., sharding the workload across multiple instances).
    ii. Data Scraping Optimization:
        - Tweak scrape_interval and selectively scrape targets to reduce unnecessary load.
    iii. Storage Optimization:
        - Adjust retention policies, optimize TSDB, or integrate remote storage solutions.
    iv. Query Performance:
        - Optimize PromQL queries, use recording rules, caching, or sampling.
    v. Alerting and HA:
        - Scale Alertmanager, manage alert deduplication, and ensure a robust HA setup.
    vi. Network and Infrastructure:
        - Optimize load balancing and network paths to handle high volume traffic.