1 Design a metrics reporting system

1.1 Question

Talk me through how you would design a system to create and report metrics for user actions and other events occurring on an Android app.
Assume the metrics will be sent to a SQL database through a REST API which you control. The backend design is out of scope.

Examples of metrics your system should support include:

- counts (e.g. button clicks)
- timers (e.g. time taken for a network call)
- aggregate statistics (e.g. data usage in a given time period)

1.2 Expected clarifying questions

- What attributes should be in each metric? What kind of queries are expected to be performed?
    - Should be able to query according to deviceId, platform version, device model
    - Should be able to query according to which screen/workflow/state the app was in when the metric was emitted
- Do the metrics need to be uploaded in real-time? What kind of delay is acceptable?
- Does the app support offline usage? If so, is there a limit to the batch size?
- Is there a minimum acceptable network strength before the metric can be uploaded?

1.3 Design considerations

- Storage/persistence
- Threading. Events can be generated in main thread but need to get uploaded in the background
- Network connectivity, backend unavailability - need a way to store the metrics
- Network/data usage
- Battery usage

1.4 Criteria

	Raises the bar (SDE II)	**Meets the bar (SDE II)**	**Lowers the bar (SDE II)**
Storage/persistence	• understanding the tradeoffs and coming up with multiple options	• Design involves persisting metrics as local File, AtomicFile or local DB	• Does not account for offline or poor

	• Picking one approach over the other. and explaining how each approach would work.	asynchronously when the metric is generated • Candidate accounts for staleness (e.g. metric persisted locally for 1 week without a successful upload ...)	connectivity cases, which necessitate storing the metric locally until they can be successfully uploaded • Proposes to store the metrics either as instance state Bundle (which does not consider complete process death) or in Shared Pref (which does not support durable disk writes)
Upload	• Without prompting, design supports cases when fill rate is occasionally higher than drain rate (i.e. metrics being generated faster than they can be uploaded)	• With some prompting, design supports occasions when fill rate is occasionally higher than drain rate by setting a max batch size	• Attempts to correct imbalance solely by increasing frequency of uploads. • Disregards issues with battery usage and network

			performanc e
Concurrency	• Leverages DB transactions to handle concurrency, without any custom locking/synchronizati on • For multiple tiers, supports a solution that uploads metrics for each tier independently of one another	• Without prompting, recognizes that metrics can be generated in UI thread but uploaded in background thread • Handles concurrency through custom locking/synchronizati on blocks • using existing ios libraries instead of creating your own	• Needs to be prompted to account for threading • Proposes to use AsyncTask (Android specific) • Design would cause main thread to freeze (App Not Responding , i.e. ANR) • Created a queue and gave it a high priority.
Network usage	• Candidate offers industry-standard solutions to minimize data usage per upload (e.g. GZIP compression, protobufs) • Distinction why they are using Protobofs vs JSON • Supports retry in the case of server failure, and also handles	• Attempts to reduce data usage by uploading metrics as a batch • Supports a retry mechanism in the case of server failures • With some prompting, differentiates between various failure responses. e.g. considers how to	• Does not support batching (i.e. design involves uploading each metric one by one) • Does not support retry in the case of

	scenarios with multiple consistent server failures • With minimal/no prompting, differentiates between various failure responses. e.g. considers how to treat HTTP 400 (BAD REQUEST) vs. HTTP 500 (SERVER UNAVAILABLE). What is the customer impact? • How do they use the bad request? • back off algorithm.	treat HTTP 400 (BAD REQUEST) vs. HTTP 500 (SERVER UNAVAILABLE)	server failures
Battery usage	• Candidate offers solutions that employ Android framework/OS support for minimizing battery usage (e.g. WorkManager, JobScheduler) • Adaptiveness- plugged in vs not plugged in • Wifi versus cellular	• Attempts to reduce battery usage by uploading metrics periodically at suitably large intervals (e.g. greater than 1min)	• Upload as soon as metric is created
Configurability	• Without prompting, candidate's solution supports multiple tiers of metrics (e.g. CRITICAL vs. NORMAL vs. DEBUG), with each tier's storage and upload policies	• Supports multiple tiers with some prompting by splitting the metrics up into multiple queues or tables.	• Naive support for multiple tiers (e.g. iterate through queue to find and

	independently configurable according to network strength, available storage etc. • Without prompting, candidate offers solutions to make the system configurable for multiple apps • setting up feedback loops	• Turn on and turn off features	upload CRITICAL metrics and upload, then iterate through same queue for NORMAL metrics, etc.)
General design	• When deciding how to account for cases where app crashes just before/after uploading metrics, candidate favors solution that minimizes 'lost' metrics • Design supports multiple Upload policies that provide unified handling of fill-rate issue, battery usage, network usage • Modularity : making library • Extensible fields	• Asking clarifying questions for metrics • using the available framework • Adding basic fields	• Confuses metrics reporting with logging, even after repeated prompts • Proposes to use an existing 3rd-party solution and is unable to explain its internals