1. When you are testing a new feature, under what circumstance would you deviate from the script while performing manual test

The first thing to take note of before testing a new feature should be to identify what all tests should and would break because of the new changes. Post that we prepare a plan for fixing the breaking tests as well as adding new tests specific to the new feature based on the timelines. Also if the new feature breaks any tests that should not have broken because of the changes, they should be filed as bugs.

Also another thing to keep in mind would be taking a call on what all tests would be ideal candidates for automation. Adding every test into the automation suite might be costly in terms of cost and maintenance. However, it is very important to identify the right set of tests and not miss any critical business flows. Also some complex scenarios that are difficult to reproduce are best done manually. For eg:

1. Adhoc / Exploratory tests to dig deeper into certain problematic areas in the code. Again we might choose to add a subset of these tests as well into the suite based on the criticality of the test.

2. Reliability/High Availability tests to ensure that different components work together properly even if some components go down for a while. But again, there are tools like chaos monkey that can be used for these tests. However, they should not be part of your regression automation suite.

2. If you joined a team of 10 developers as the only QA engineer and could implement and change any processes you like, how would you ensure that the team delivers high quality software?

As the only QA engineer in the team I would have to first educate the developers on the importance of Test-Driven development. Quality of the product is a joint effort and developers have to work equally towards ensuring that their code is of the highest quality. Also being the only QA engineer in the team, with 10 developers working parallely on features, it looks highly unlikely that all the features can go through an E2E QA cycle and the QA engineer. Hence I would try and implement the following steps:

1. Ensure that every feature is discussed in the detail and risks called out early.

2. Take up the responsibility for critical features cross-cutting multiple components and requiring complex integration tests. Let the developers handle single component feature changes. Again this can always be shuffled based on the timelines and workload.

3. Ensure that every change that goes into the master branch gets code-reviewed by at least two other developers and the QA engineer.

4. Integrate code coverage metrics for all the repositories and enforce a minimum coverage [say 70% C0, C1] for check-ins.

5. Also provide test suites with comprehensive set of smoke/regressions tests which the developers can run to ensure that they have not broken any existing tests because of their feature changes.

6. Provide CI/CD pipelines for different tasks through Jenkins so that all the processes are streamlined.

3. A developer closes a defect you have reported as "Wont fix". How would you react?

Initially, I would try to talk to the developers and try to bring a discipline in terms of adding relevant details and comments in the bugs. So if the defect was returned as "Won't fix" without a reason, I would go and talk to the developer to understand the reason behind his decision. Bug triages with relevant stakeholders - Product, test manager, dev managers - is something I would try and schedule frequently. Sometimes there are bugs that are not really critical but you just open them while going all Adhoc. If that is not the case and if I feel the bug might have serious repercussions in production, I will add my comments in the defect and put my points forward in the bug triage meeting. If all the stakeholders are aligned on the decision being taken, I would close the defect with the details of the discussion for reference. In the worst case, if at all the issue occurs in production, we will have a record with all the required details for reference.

4. Provide examples of where you have personally:

a. **Attempted to convince someone to change the way they were working, even though they dint want to.**

   As part of one of the complex features cross-cutting across multiple teams, we had a few developers in our team who would not follow test-driven development. They would run unit-tests and would hand it over to the QA without running the basic dev integration tests [say integration with UI / dependent services etc.] Even the contracts between teams were not tested. I tried to explain that at least a dev integration has to be run before handing it to QA and P0's should be ironed out as early in the cycle as possible. Eventually we discussed the matter in one of the forums and everyone agreed that the basic integration should be tested and that quality is a joint responsibility.

b. **Identified a problem in the process and implemented a change to improve it.**

   When I initially joined the team of 12 developers as the only SDET, the team was in a lot of flux in terms of the processes. The developers would push changes into the master branch and break things in production and revert it back. Since Flipkart was in a startup mode earlier, people had got into this habit of doing things fast but not always right. When I joined the team, a new development manager also joined from Yahoo and we both agreed that we had to change the way things were working. I brought in the following processes to push the developers towards TDD and bring a sense of responsibility for the quality of code they build.

- Made code reviews before check-ins mandatory. The plan was to actually integrate with code review tools like Gerrit to enforce the habit of code reviews. But eventually we dint need the tool since we saw that the developers understood the importance of code reviews and started following it diligently.

- Integrated code coverage tools with the repositories to ensure that the unit tests are written and have a certain coverage (say 70% CO, C1)

- Identified the important smoke tests and provided the suite to developers so that they run the basic smoke suite on their local and ensure their changes have not broken existing tests before pushing into master.

- Built an E2E Jenkins pipeline so that continuous integration and deployment is adopted and developers don’t push changes directly to the master branch. The jobs in the pipeline were – run the unit tests and get the code coverage metrics, trigger the downstream build & deploy job if the basic coverage % was met, and finally run the smokes and regression suites on the hosts.

- We also integrated Jira with Git so that every commit or push is associated with a Jira for reference of changes going in.

c. Implemented new testing tools or frameworks

- **Mock framework**: The team that I was working for had around 7-8 services, and we had a lot of external dependencies on other micro-services. Some of the downstream dependencies were not user-facing services and as such they would not care about maintaining the stability of their systems. Maintaining the nightly smoke/regression suites became an issue since the tests would break mostly because of downstream system issues. I worked on building a mock solution for the systems in order to shield the daily smokes and regressions from flaky dependent systems. The same mock framework was also used for mocking out the dependencies during performance runs for benchmarking our API's. The mock service was written using Dropwizard framework and had redis data store in the backend for storing the requests and mocked responses.

- **Distributed tracing**: Since ours was a microservice architecture, there were lot of microservices involved and as such it was difficult to track down a certain failure to the exact downstream system through the logs. I worked on setting up distributed tracing for the services using an open source solution called Zipkin. I set up the Zipkin components and Kafka cluster and also instrumented the required changes in the Java services to push the tracing information to Zipkin collectors.

d. Made changes to improve the quality of code before it reaches testing stage

- There was a certain feature which was heavily dependent on the downstream responses. We were the seller facing team which was supposed to present a view of the seller's products in the warehouse. The warehouse software was maintained by a separate team which would update the status in the system for every physical movement of the product inside the Warehouse. During the design discussion for this feature, there were lot of confusions over what all states must be exposed to the seller and during the process I also realized that a lot of states could get missed because of this. Also making an API call for getting the results every time was not a viable solution since the responses were huge and the warehouse team could not make the response paginated due to some constraints. I suggested that since the seller’s products are warehouse constructs, the warehouse teams should be owning that data, but we could build a view on our end which we could update based on events from the warehouse. Even though the stakeholders thought it was a good idea, because of timeline constraints we went ahead with the initial approach and decided that we will pick up the task to build a view in 6-9 months’ time. Eventually there was a full-fledged feature around the suggestion and is currently implemented in our system.

- Another instance I can think of is for a relatively isolated feature in terms of dependencies. The system was supposed to collect numbers from 2 separate stores, make some complex math on them and finally respond to the end user with a YES or NO based on the result. When I went through the design and code for this change, I realized that there was no way for us to actually test the feature correctly because the API would just respond with a Boolean, but we will not be able to test if the actual calculation within is correct. So I suggested that we either build test API's for automation, or add sufficient unit tests to check the results of calculation for different use cases. I also got sufficient loggers added so that we display the exact calculations on the console for testing purposes.

- As part of the performance benchmarking for API's before the ‘big-billion day’ sale, even before running the actual benchmarks, I noticed that some API's had high latencies, and the logic on our end was not too time consuming, the call to downstream system was the one taking more time. And this was some information we required to validate with downstream system before proceeding with our business logic and this information was usually not too dynamic in nature. I opened a JIRA for introducing a caching layer for caching the information rather than make API calls for every operation. Later the benchmarks revealed that introducing a caching layer gave a huge performance boost to a lot of our API's

e. **Recognized a pattern of bugs occurring (e.g. cross-browser rendering problems) and implemented a change to stop the problem.**

   - The caching implementation mentioned above was one such change. Lot of our API's had failures in production because of downstream failures. We abstracted ourselves out by adding a caching layer.

   - In our automation framework, some use-cases required checking the status change on a different service which gets called asynchronously. When we call the required function and assert the status change, the tests used to fail sometimes since the asynchronous calls would take some time to reach through the Kafka cluster. I implemented a wait helper to keep checking the status at regular intervals until a timeout period after which the operation would fail.

f. **Any other achievement in your career that illustrates your suitability for our Quality Assistance role.**

   In my 8 years as a Quality Engineer, I have raised testing standards by experimenting with new processes and adopting new tools. I believe in understanding the product/system as a whole and breaking down problems into components and solving any areas with room for improvement. I have been awarded the SPOT award for taking calculated risks and delivering products meeting quality bar. I have also been self-driven. I am interested in different languages, frameworks and approaches which has helped me improve our culture and tools at Flipkart. I have also been awarded the Kudos award in Flipkart. I believe I have good communication skills, willingness to be able to talk to the business, defend their interests and work with the developer to meet those needs. I am also passionate about Quality and I feel that with my overall experience I will be a good match for the job description you have mentioned for the role of a Quality Engineer.