Designing Performance Testing Software for Cloud Based Enterprise Applications

Pranshu Mishra
pranshu188@gmail.com
Surbhi Agarwal
surbhiagarwal975@gmail.com
(Dated: *February 26, 2024*)

The paper presents the design and implementation of performance testing software designed for cloud-based enterprise applications. With the increasing adoption of cloud computing in enterprise environments, ensuring the performance and scalability of these cloud-based applications is crucial. Traditional performance testing tools may not fully address the unique challenges posed by cloud environments as these tools are, generally speaking, usually designed as load generators. Therefore, we propose a comprehensive approach to designing performance testing software specifically for cloud-based enterprise applications in which load generation is one component of it's architecture.

I. INTRODUCTION

Contemporary performance testing tools predominantly prioritize load generation, often sidelining the broader scope of performance engineering duties. From a performance engineering perspective, load generation represents merely one facet of a multifaceted responsibility. A holistic approach necessitates thorough post-load analysis. This paper advocates for a paradigm shift towards a "performance-first" ethos, fostering a culture of iterative refinement grounded in data-driven analytic. Our aim is to explicate the foundational tenets guiding the development of a framework conducive to good performance testing practices. By integrating elements such as a rigorous post-load scrutiny, a streamlined reporting and notification mechanism, our endeavour is to cultivate a robust performance testing paradigm tailored to cloud-based enterprise applications.

II. ASSUMPTIONS

A. Target Application

Since we are designing a performance testing tool that generates load, it's essential we clearly define who the target is. We'll be designing our software keeping in mind these assumptions:

- The target is a cloud based application with multiple domains/modules.
- It is a multi-tenant multi-user system or even a multi-user system with no tenancy.
- The target is implemented as microservices architecturally.
- The target's components are deployed as containers, using a container orchestrator tool like Kubernetes.
- The different domains/modules within the application have dedicated team(s) per module.
- The teams can deploy their changes independently of each other.
- The deployments first happen in dev/staging environments first before going into production.
- Just like production and dev/staging environments, there is a separate testing environment available that matches production's deployment. This environment will be the actual target for our performance tests.

B. Our hypothetical performance testing software

We will designing our performance testing software keeping in mind these assumptions:

- The performance tool will not be used more than once a day. This ensures we have a single complete run for all the performance test cases and leave enough time for result analysis.
- Performance tests will be executed everyday. A daily report and a weekly/month summary report will be produced by the tool.
- Teams will keep adding new/updating existing performance tests whenever they plan to release changes into production.

III. SYSTEM DESIGN

A. Components

These are the logical components that make up the proposed performance testing tool:

Test suite repository: This component holds the tests that need to be executed by the performance testing tool. This can be implemented either as a monorepo or separate repos each owned by a team/module.

Orchestrator: This component loads the test suite and processes it for execution. It generates a test plan

and forwards it to the load generator to put load on the target system.

Load generator: This component is the only part of the performance testing tool that interacts with the target. It receives instructions from the orchestrator and executes upon them.

Reporting: This component generates a report that the user can view to understand the target system's behavior.

Notifications: This component sends out alerts/messages via specified channels regarding important events or actions. Examples include test plan execution start, test plan execution end, faulty tests, SLA violations, failure limit violations, etc.

Database: This component is used to persist test plan execution data. This can include derived metrics.

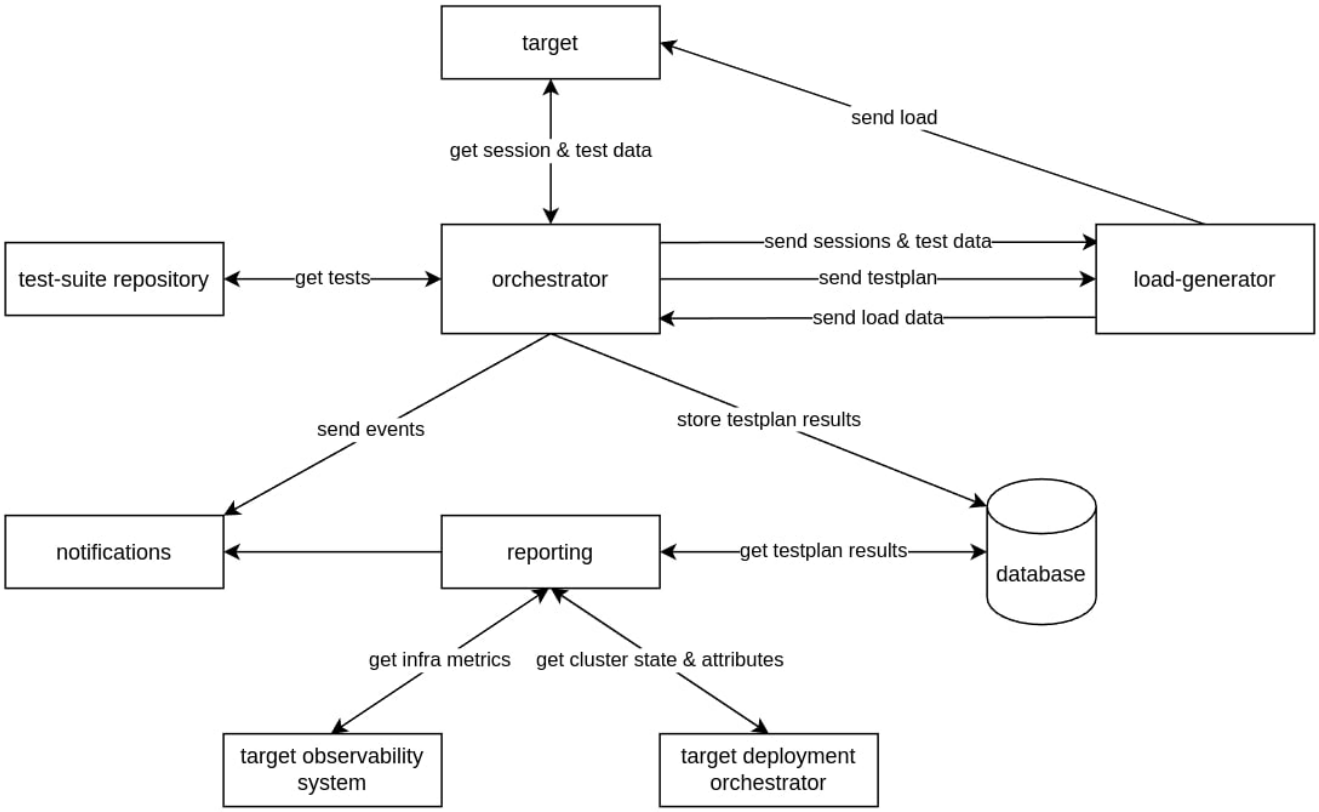

FIG. 1: **Component Interaction**

B. Implementation

For the implementation, the orchestrator, reporting and notification components are all housed in a single binary (referred in the diagram as 'core') while there are multiple instances of the load-generator that's deployed on separate hosts.

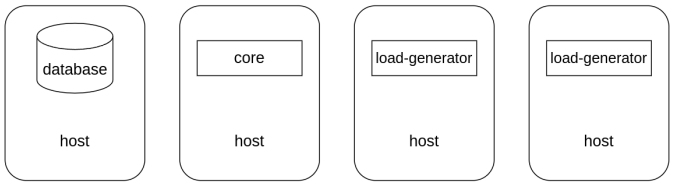

FIG. 2: **Host deployment**

C. Test Plan Execution

The performance testing software's core operation can be broadly categorized into 3 phases: Initialization, Execution and Reporting.

1. *Initialization Phase*

Target Setup
Since we are going to conduct a performance test, it is important to match the target's setup with production. The data kept in our target's databases doesn't have to be real data from production, but a close enough representation is a very good point to start from. As part of the setup, we want to scale the target to match production, i.e., this would include scaling the number of hosts and/or containers within the target. Since the performance tests that would be executed may produce additional data in the target storage system, it's a good time to reset the target's data from a safe backup now.

Session Generation
Enterprise multi-tenant applications typically require their users to authenticate & create a session. If the session is long lasting in the production environment, it makes sense to mimic the same behaviour during the testplan execution as well. The performance tool can log in to the target beforehand so as to create & persist session(s). These session IDs/tokens/strings/etc can be shared with the load generator for usage during the execution phase.

Test Data Pre-Caching
Some tests may target certain APIs (typically HTTP GET) that require some data to be given to them, for example, GET /api/info/v1/users?id=123. In this example, the user with ID 123 must exist in the target system else the API returns an error message. A typical test would be written to call a different API endpoint to fetch all user IDs and then use those IDs to generate load, but doing this in-between the testplan execution is not ideal as it introduces side-effects by making extra calls to the target system. A good way to deal with the problem is to fetch this static data beforehand & persist it. The persisted data can then be later on used during the testplan execution phase, thereby removing the need to make extra API calls.

Testplan Generation
This is the stage where the performance testing tool will determine what tests will be executed by the load-generators. An execution testplan is generated which contains step-by-step instructions on what the load-generators are supposed to target & when, as defined by the test authors. This execution testplan is then handed-off to the load-generators.

2. *Execution Phase*

In this phase, the load-generators get the testplan and begin the execution of it. If deployed as a distributed configuration, the load-generators clocks must be in-sync before the test starts so that all the load-generators can start the testplan execution at the same time.

3. *Reporting Phase*

Clean up: While we have considered database resetting before testplan execution for our target, however, not all parts of the target system's persisted data maybe be backed-up or reset given the design of the target. If the target permits to revert the state that the performance tests have changed, those APIs must be used. The testing environment must also be scaled back / destroyed as it won't be used until the next testplan execution.

Report generation: Our tool will now generate a report for us to see & let us know how the testplan execution went. The tool will also fetch infrastructure level metrics from the target's observability system.

Notifications: The tool will now send out notifications via specified notification channels (like email) regarding certain events like SLA violations, failure limit violations, faulty test executions, and high level testplan execution summary (like executed tests and module-wise breakdown).

RCA ticket creation: If configured, the tool may create automated follow-up RCA tickets in the company's internal ticketing system (like Atlassian JIRA) for events like faulty test executions or SLA violations.

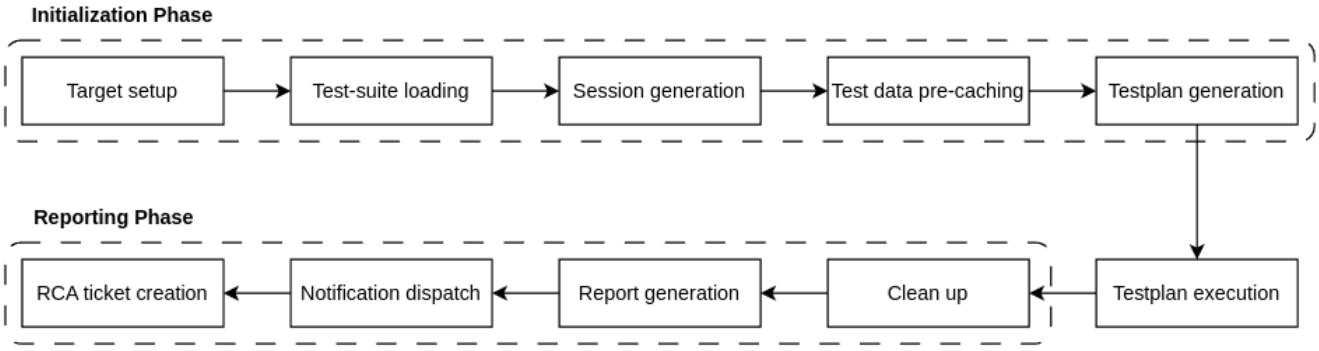

FIG. 3: **Test Plan**

IV. OPERATIONAL OVERVIEW: USING THE PROPOSED SOFTWARE

With all of the points implemented above, this is what a typical routine operational workflow of a user of the proposed system would look like:

1. Teams add new performance tests
2. Daily test triggers
   (a) Environment is scaled to match production
   (b) Relevant databases are reset and restored from backups
3. Test plan is executed
   (a) All performance tests are triggered one by one as dictated in the testplan
4. Post test plan execution, report is generated
   (a) Environment is scaled back / destroyed
5. Based on the report, notifications are sent to relevant test authors / owners
   (a) Notifications on faulty test executions
   (b) Notifications on Service Level Agreement limit violations
   (c) Notifications on failure limit violations
6. Report data is persisted in database
7. Automated RCA support tickets are generated in case of faulty test executions & tagged to relevant test author
   (a) Test authors fix defective tests
   (b) Test authors close RCA ticket
8. Daily test executions are conducted for a week / month
9. Summary report is generated for the past week / month using report data in database
10. Automated RCA support tickets are generated in case of SLA violations, failure limit violations
11. Test authors use relevant observability systems to analyze test results & tweak the SLA/failure limits
12. Test authors close the RCA support ticket

V. CONCLUSION

The workflow proposed herein transcends the traditional role of a load generator. It serves as a pivotal workflow enforcement mechanism, instilling a practice of prioritizing application performance throughout the software development life cycle. Its inherent capability to retain historical test results and target configurations enables users to identify areas of system degradation effectively.

VI. FUTHER IMPROVEMENTS

1. Introduce chaos engineering by causing container / host outages mid-way during testplan executions
2. Introduce machine learning to predict performance trends and understand what changes in the target system cause changes in trend-line

VII. ACKNOWLEDGEMENTS

1. Kleppmann, M. (2017). Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. O'Reilly Media.
2. Gregg, B. (2020). Systems performance. Prentice Hall.