DeepSource: Distributed Deep Health check system

About Me:

<table>
<tr><td>Name</td><td>Saahil Ali</td></tr>
<tr><td></td><td><table>
<tr><td>Github</td><td>programmer290399</td></tr>
<tr><td>Gitter</td><td>@programmer290399</td></tr>
<tr><td>LinkedIn</td><td>Profile Link</td></tr>
<tr><td>Discord</td><td>programmer290399#9697</td></tr>
</table></td></tr>
<tr><td>Contact Information</td><td><table>
<tr><td>Email</td><td>programmer290399@gmail.com</td></tr>
<tr><td>Contact no.</td><td>+91-9981789723</td></tr>
</table></td></tr>
<tr><td>Resume</td><td>Link</td></tr>
</table>

- I am currently in the pre-final year of my 5 yr M.Tech + B.Tech dual degree program in IT,
- I have *3+ years of internship experience* with various startups which you can see on my LinkedIn profile.
- I participated in Google Summer of Code 2021, under Python Software Foundation.
- My top two skills are Googling & Reading the Docs.

How we can approach this:

Identifying the Problems with Implementation:

There are many problems we need to deal with while implementing these deep health checks, I have mentioned major ones below:

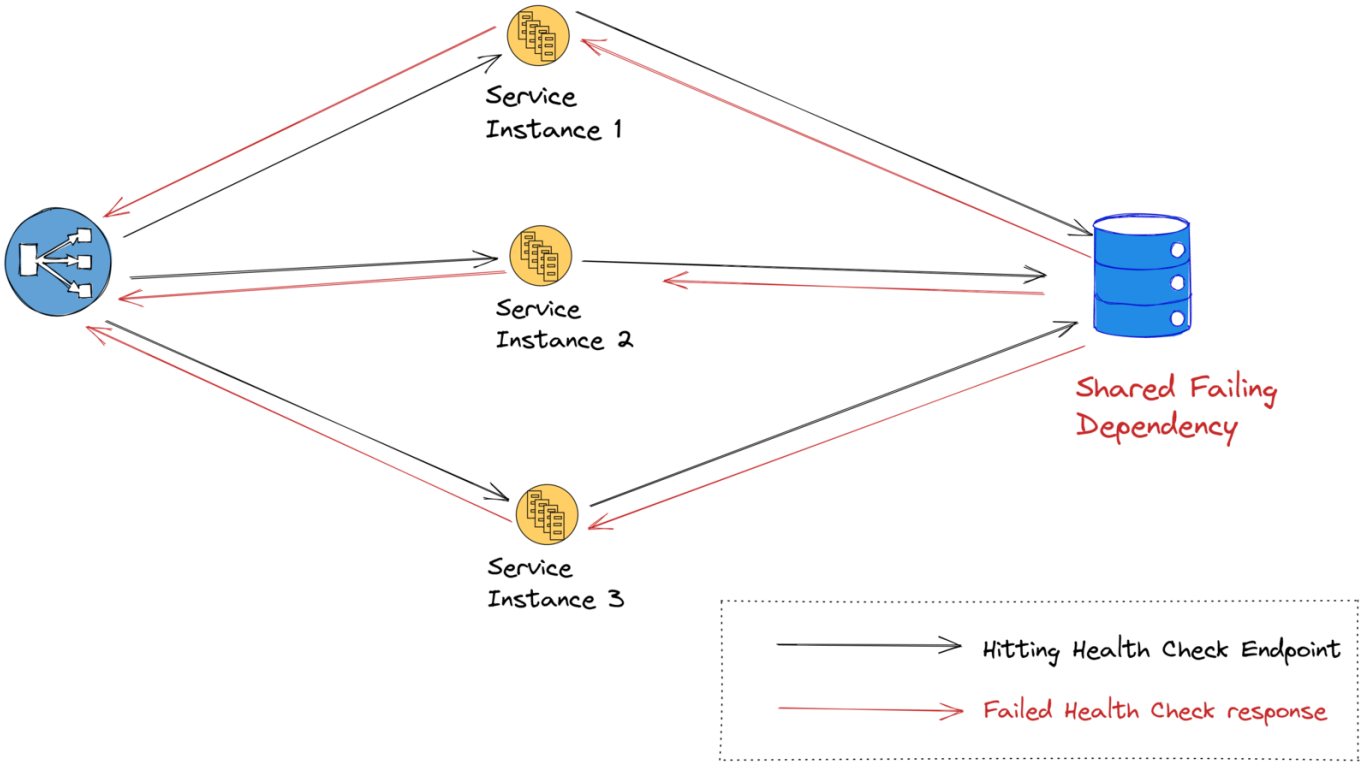
Problem 1: Keeping the Checks Light

- Health Checks if not designed properly can take a toll on our infrastructure
- For example: If we want to do a health check on a database we should make a separate table for this purpose with a few records in the target database, instead of trying to run a query on a table with millions of records, this will be light on the database and better for our system as a whole.
- Deep health checks can increase startup latency and be a big issue for container orchestration, thus they shouldn't be run when starting a service.
- One way of speeding dependency checks over a network could be to use multi-threading for network calls.
- To make sure the health checks don't overwhelm our deployments we need to carefully choose the following four parameters:
    a. How long should we wait before we send our first health check
    b. What should be the interval between two consecutive health checks
    c. How many times a service should fail/pass a check to be considered unhealthy/healthy
    d. How long shall we wait for a response on a health check

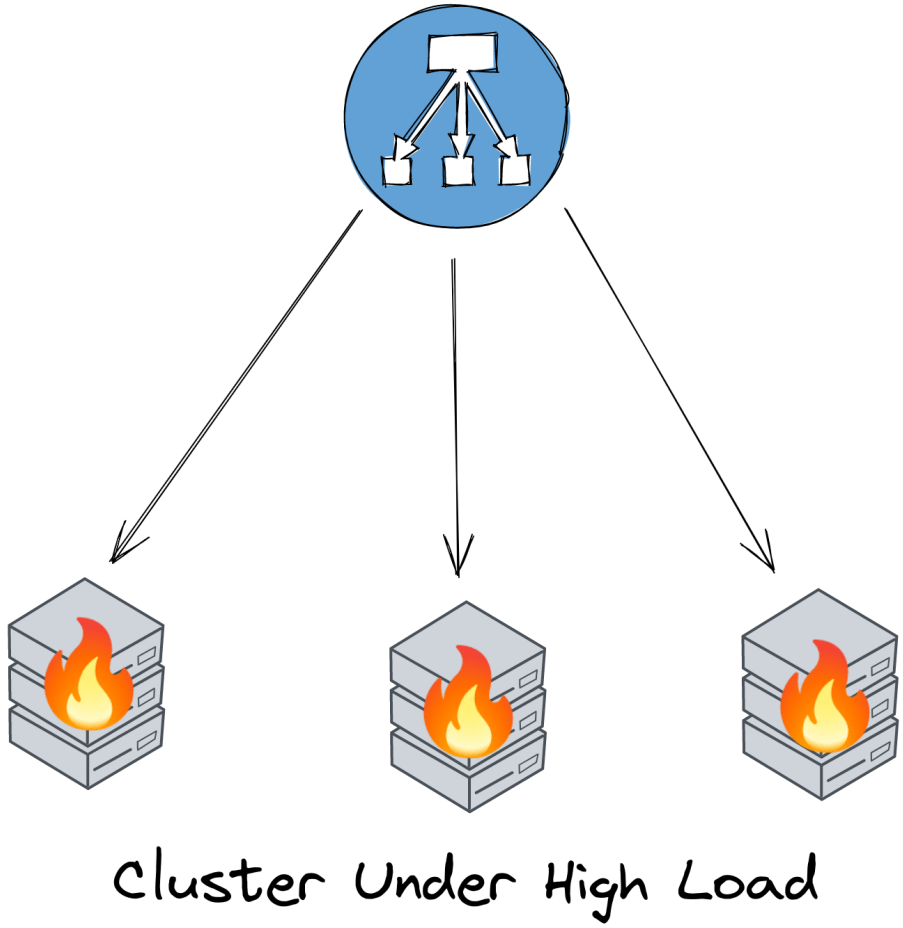
Problem 2: Preventing Cascading Failures

Causes:

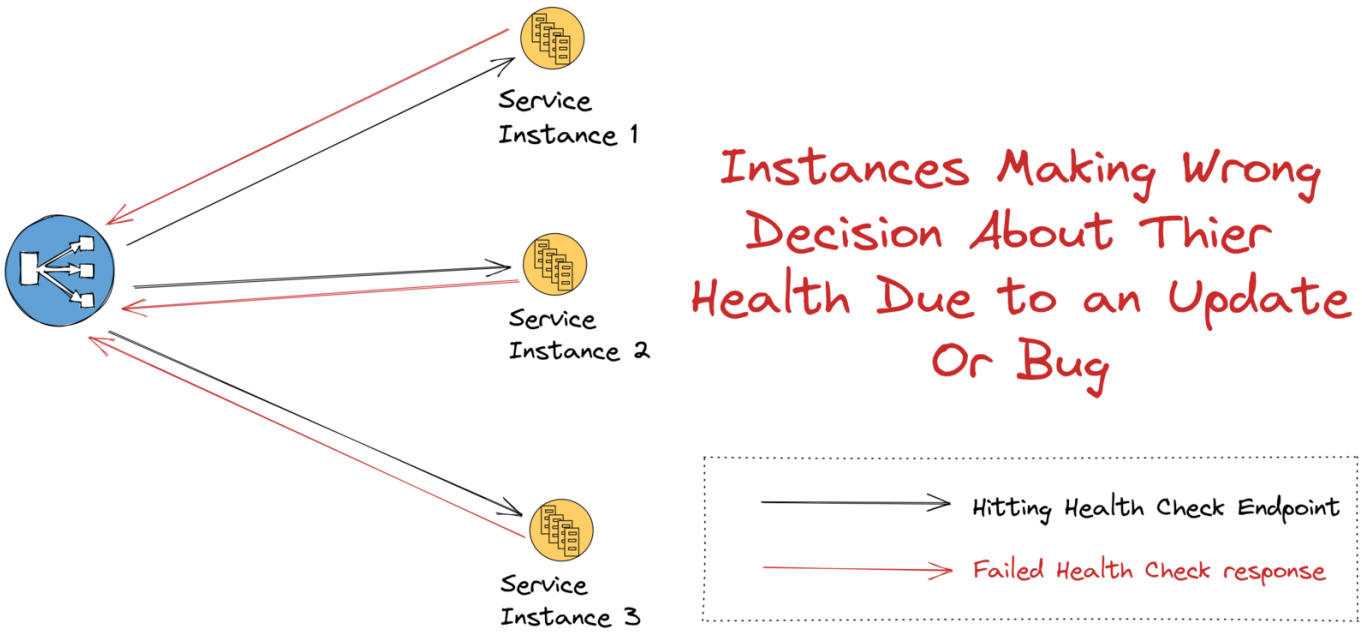
1. Failure of a shared dependency



2. High Load on the Cluster



3. False-negative checks

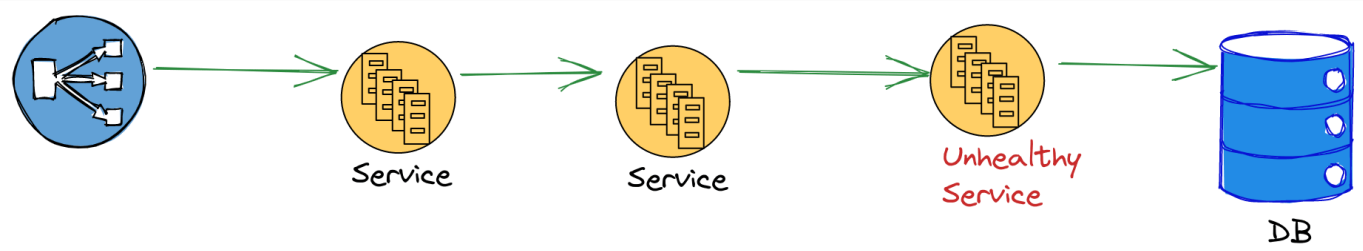


Possible Solutions:

1. Most load balancers have the ability to detect cascading failures and in such cases they'd open traffic to all hosts, ignoring these checks to prevent a massive outage.
2. A better way to approach this is to monitor the metrics of a cluster as a whole and find outliers and then remove them from service rather than blindly using health checks. If under load all servers will be having a similar situation and thus none of them should be taken out if their response is too slow for health checks
3. Other than that having mechanism that has more granularity in performing health checks can help identity which part of the system is failing and due to that which others will be reported as unhealthy.
4. I've worked on such systems in the past that can reliably perform anomaly detection on the stream of metrics and then perform root cause analysis to find the actual issue at cliff.ai.

Problem 3: Reporting False Positives

- Deep health checks can have this domino effect where we check on a service that checks on another service and so on and some service in this chain is failing in such a case, this failure will cause a failure on all the upstream health checks.



- This kind of issue can be mitigated by having more granular information on services and their dependencies as discussed above as well, and by marking certain dependencies as critical and non-critical, to reduce their effect of it.

The Proposed Architecture:

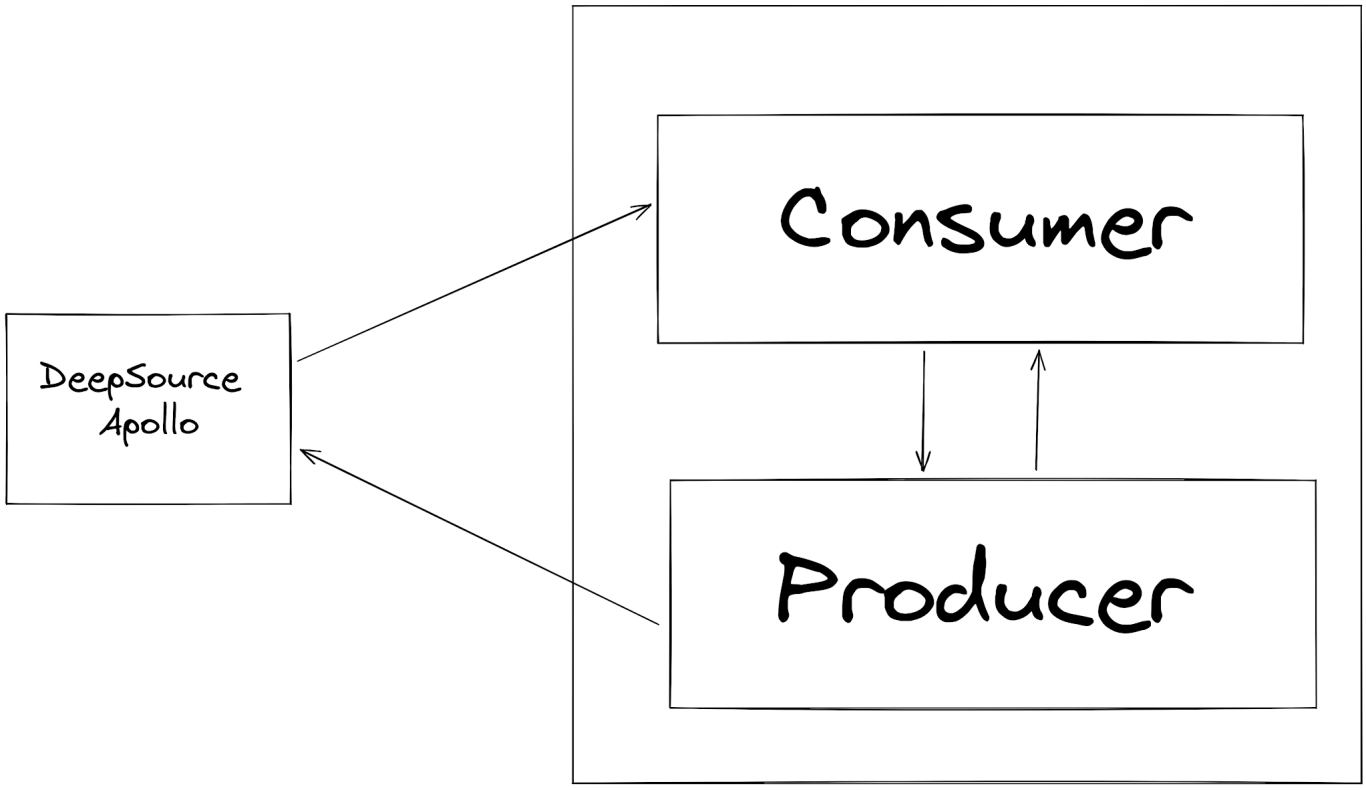
- Taking a look at the problem from a higher level, we should prefer building it in as modular & extensible way as possible, thus here I am dividing the services in our system into two groups:
  a. **Consumers:** Services that need to check on the health status of other services
  b. **Producers:** Services on which we need to run health checks

- We have three major types of consumers:
  1. **Load Balancers:** Need to monitor services
  2. **Container Orchestrators** (Like Kubernetes): Need to monitor containers
  3. **Monitoring Systems** (Like AWS Cloud Watch): Need to monitor daemons, background jobs, etc

- We have many types of producers like databases, running app instances, daemons, web servers, etc.
- Now as the project page says we shall make plugins for all the producers so that we can perform the following checks on them:
  - **Local Health Checks:**
    - These health checks test resources that are not shared with the server's peers. Therefore, they are unlikely to fail on many servers in the fleet simultaneously**.**
    - For example RAM, CPU & Disk usage, OS-related services like daemons, etc.

  - **Dependency Health Checks:**
    - These health checks find out if the service can interact with its dependencies or not.
    - For example, the ability to connect to a DB or other services it might need to serve requests.

- A high-level view of the proposed architecture has been shown below:



- Each Producer runs with a plugin attached to it so that it can interact with our consumers and/or our main health check system, Apollo.
- We can further classify the producers into three types:
    - **Load Balancer Facing Producer**
    - **Container Orchestrator Facing Producer**
    - **Miscellaneous Producer**
- Apollo integrates with the consumer to keep the consumer updated with the latest health status of various producers, it maintains all this data with timestamps in an appropriate database.
- Apollo polls all the producers as per the defined schedule for health checks.
- Apollo keeps the track of all the dependencies of various producers using a dependency graph.
- This dependency graph can be used to address various problems we've discussed above.
- The local health check data collected by Apollo can be used to spot outliers in a cluster and take the necessary action.
- In case we don't want Apollo to be in between our producer and consumers, our plugins must be designed in a way that the consumers can directly fetch the required information from our producers, however, this is not ideal in my opinion.
- Some projects from which we can draw inspiration are listed below:
    - **Amon:** This project is pretty similar to Apollo, they also have a plugin-based architecture
    - **Goss:** This project is a very good inspiration for plugins that we'd need for miscellaneous producers like daemons.
    - **Runitor:** This can be seen as a good example for remotely executing tests of various servers
    - **Healthcheck:** A good example for Kubernetes integration in Apollo.
    - **Django-watchman:** Exposes a status endpoint for backing services like databases, caches, etc.

Project Timeline:

Week No.	Goal	End-Result
1	Discussing the proposed architecture with mentors, taking a deep dive into the pre-existing source code. Finalizing which all plugins should be made during the externship period (Ideally 4).	Getting Started with the project.
2	Starting to work on Apollo, implementing a basic working model based on the mentor's input.	Getting started with the development of Apollo
3	Finalizing Apollo with all documentation, and CI/CD pipelines.	Completing Apollo's basic functionalities.
4	Discussing and implementing Plugin 1 with mentor's input	Getting Started with the development of Plugin 1
5	Finalizing Plugin 1 with all documentations, and CI/CD pipelines.	Completing Plugin 1 development
6	Discussing and implementing Plugin 2 with mentor's input	Getting Started with the development of Plugin 2
7	Finalizing Plugin 2 with all documentations, and CI/CD pipelines.	Completing Plugin 2 development
8	Discussing and implementing Plugin 3 with mentor's input	Getting Started with the development of Plugin 3
9	Finalizing Plugin 3 with all documentations, and CI/CD pipelines.	Completing Plugin 3 development
10	Discussing and implementing Plugin 4 with mentor's input	Getting Started with the development of Plugin 4
11	Finalizing Plugin 4 with all documentations, and CI/CD pipelines.	Completing Plugin 4 development
12	Wrapping up the complete project with any remaining work related to documentation, changes, etc.	Successful Completion of the Project

NOTE: This is a tentative timeline that is subject to change depending on the mentor's input

Projects I've worked on:

Project Name	**Type**	**Link**	**Tools & Technologies Used**	**Description**
PyQnA	Personal Open Source	GitHub Repo	Python, Git, Github, Shell scripting	• PyQnA is a simple python package that aims to provide a consistent and unified API for all Question Answering related tasks in Python. • It is available via PyPI, I've made 3 releases so far. • It is still in the Alpha stage. The architecture of the package is modular, i.e. users can install specific parts of the package as per requirement. • This project also participated in **HacktoberFest 2021**. And helped new contributors start their open-source journey.[1] [2]
DFFML	Community Open Source	Github Repo	Python, GitHub, JavaScript, HTTP APIs, MLOps, Git	• I worked on this project during my Google Summer of Code 2021. • Since then I have been actively contributing to this project and am among the **Top 3 contributors**. • I have made more than 25 PRs to this project and have opened and worked on many issues as well. • Also, I actively help new users and contributors in getting started with DFFML.
Cliff.ai	SaaS Product Proprietary	Product Page	Python, FastAPI, Cython, AWS Lambda, ReactJS, Shell Scripting, Docker, GitLab, Git	• I worked on this project during my Internship at Valyrian Labs Pvt. Ltd. • I worked on the development and optimization of various anomaly and outlier detection algorithms. • Optimized the core anomaly detection algorithm, which *translated into **27% lower costs** for cliff.ai's customers.* • Helped DevOps team to scale the various algorithms and deployed them as AWS Lambda functions.

Greendeck.co	SaaS Product Proprietary	Product Page Product Crunch -base Page	Python, FastAPI, AWS Lambda, ReactJS, Shell Scripting, Docker, GitLab, Git	• I worked on this project during my Internship at Valyrian Labs Pvt. Ltd. • I worked on the development and optimization of various product matching algorithms. • Designed a text classification pipeline to process millions of records within minutes, leveraging GNU tools, Python, Bash, multithreading. ***Reducing the cost of computation by almost 50%.***

NOTE: I have worked on many other open-source as well as proprietary projects, most of them can be seen on my GitHub profile, I have only shared a few major ones up here.

What I expect to learn at DeepSource:

- How to design systems that can work reliably at such a great scale.
- How the System Design theory is actually put into practice. How to deal directly with millions of users.
- I want to learn more about the software development methodology at DeepSource.
- I also want to polish my software development skills by working under the guidance of DeepSource engineers.