Microservice & DevOps

The Warehouse that Blinks

Welcome back.

FrostByte Logistics was highly impressed with your work on the network validation system and now values your expertise even more, and is confident to present you with more complex tasks of the industry.

One of their partner companies, Valerix, operates a massive e-commerce platform, processes thousands of orders per minute, and must deliver both deadline agreements and correct orders.

On a busy Tuesday morning, their team sits down with the FrostByte systems architect to discuss a major problem. After a while, you are called in and presented with the challenge -

Currently, their system is a single monolithic server. It handles everything - taking orders, tracking inventory, updating shipments, and notifying users.

It works… most of the time. But there are significant issues:

- If inventory updates slow down, the entire order flow stalls.
- Adding new features is risky because all components are tightly coupled.
- Partial failures, such as database locks or network delays, can cascade into long delays for users.

You notice that the platform services are built on a poorly designed database and a heavily accumulated legacy codebase.

All together, the system is ill-equipped to handle sudden traffic spikes, and the development team struggles to safely introduce new features or scale the platform to support increasing demand.

Hence, you decide to break this monolith into microservices.

Your system architect gives you the freedom of designing the appropriate database schema and the microservice architecture for the new server.

However, he insists that, in any case, the following two services should be included as separate services in your design to maintain a necessary and proper logical boundary.

- Order service – receives orders, validates them, and coordinates downstream processes.
- Inventory service – manages stock levels and updates them when orders are shipped.

The Order service will call the Inventory service to update inventory whenever an order is ready to ship.

The Vanishing Response

Designing a robust microservice architecture is no easy feat, but a team like yours must not just stop here, and must not be careless about the reliability of a service you will provide.

So think about this -

In the real world, systems are messy.

Networks fail, services start, or restart later than expected, and there can always be noisy neighbours (Yes, they are everywhere).

To try and tackle these issues, first, we will simulate them by introducing a "**gremlin Latency**" - that is, your Inventory Service will sometimes delay its response by several seconds.

This is how it will work -

- The Inventory service should sometimes show significant latency in a predictable, deterministic pattern while responding to the Order Service.
- The Order service should continue to work smoothly even when this happens. It should not keep waiting forever for a slow response.

If the Inventory service takes too long to reply, the Order service should stop waiting and return a clear appropriate message to the user.

This was not part of the monolith but is now introduced to test the resilience of your new architecture.

Your goal is to ensure that the Order Service can handle these delays gracefully, returning timeouts or user-friendly error messages instead of freezing.

Beyond that, the system should be modular, allowing it to add services like notifications, payments, or analytics without breaking the core flow, for the Valerix Engineers.

It Runs On My Machine

Of course, you know that your work is not done.

Maybe your system has the most robust architecture and the best reliable database design in the world, but you can't say to someone - "it runs on my machine" when it fails in a real-world scenario. Your system has to survive network failures, or a localized blackout while being hammered by thousands of desperate online buyers trying to get that flash sale.

So you need to verify your escape route, and you need to automate the verification, so that nothing goes past your filter before it goes out wild in the world.

Your pipeline should be able to start the system automatically and run a series of requests against the Order Service to verify its behavior under load.

During this process, slow responses should occur in the Inventory Service and the orders affected (if any) should be recorded in a clear way, without interrupting the overall test flow.

Remember, even when some requests are affected by delays, the remaining requests should continue to run and complete normally.

Go Beyond Your Logs

After completing the fully workable server, and automating the verification of your Order Service working in a stressful environment, your code is now ready to go into production.

Or is it?

So far, for any issues you have faced while building the backend, you have managed to detect efficiently scrutinizing the error logs in your system.

But logging "Error" to a terminal is useless when you have a hundred nodes of your server running, and millions of users hitting their APIs frequently. It will not be plausible to monitor the server efficiently. Rather, we will monitor the experience first.

For that, we will need sophisticated health checking and monitoring dashboards.

Each of your services must have a **/health** endpoint. This isn't just 200 OK.
The health check must verify its downstream dependencies.
For example, If the Inventory Service cannot ping its own database/tables, the **/health** endpoint must return **an appropriate error message**.

You also need to set up a **visual alert**. If the average response time of the Order Service exceeds 1 second over a rolling 30 second window, a component on your dashboard must change from green to red.

A Schrödinger’s Warehouse

Just about when you are happy with all the hard work you have put in so far, your system architect says - stop.

He introduces you to a very significant possibility of your server behaviour, when it goes into the production, **a ghost in your server**.

You’ve handled service latency, and you’ve handled service monitoring.
But now, you must handle **partial success** cases your server can arise in the production.

Welcome to the **Schrödinger's Warehouse -**

Consider this regular scenario. Your client platform receives an order to buy gaming consoles. Your Order Service handles the process and when the order is shipped, it tells the Inventory Service to appropriately adjust the stock.

In a high stress environment, and with a poor network infrastructure, even with these regular orders, your server starts to misbehave.

What might happen, is that -

- Immediately *after* the database commit, but *before* the HTTP response is sent back to the Order service, the Inventory Service process crashes.

- The server might have processed the order successfully, but before it could respond to the customer, a noise signal might come into the way, and return an Internal Server Error to the client.

And what it might cause - is that now client and server are aware of two different states of the situation.

This uncertain “quantum inventory”, is diagnosed as a severe headache by your system architect, and you are required to build a solution in your server that will work around these network issues, or sudden internal service crashes, and **do exactly what it is supposed to do**.

The system architect gives you a final reminder: your solution must not only address the unreliable service, but also definitively demonstrate increased end-user reliability. The most effective way to prove a system's reliability is to simulate a challenging environment and observe its performance under pressure!

So now, it is time for you to put your system under pressure, and test your own product -

Simulate the uncertain behaviors described above in your own setup, so that the services no longer behave perfectly all the time.

Introduce moments where things do not go as planned, and observe how your developed solution reacts when part(s) of the workflow suddenly becomes unreliable.

Just A Human Window

So far, you have been observing your system mostly through logs, dashboards, and automated tests. But real systems are ultimately judged by how they feel to the people using them.

So just to complete the picture, build a minimal user interface that talks to your services and simulates the functionalities. **Keep It Stupid Simple**. The goal is simply to make the behavior of your backend visible.

The First Cloud Frontier

After putting your services together, you now have a solid base of a robust e-commerce platform that can handle uncertain situations.

Now, you are challenged to **deploy your services at a small scale on a cloud provider.** Nothing massive - just enough to see your microservices live outside your local environment.

Although…The Need to Leave a Trail Behind…

You've been working tirelessly, stitching services together and managing controlled chaos, but a new thought keeps nagging at you - what if, one of our database storages crashes… and the data is gone for good?"

After all that effort, you're not about to let a single crash erase months of hard work. You want to back up your data, **multiple times** just to be safe.

But Valerix then drops the curveball. They tell you -

> "Valerix still relies on a very old-school backup service. It only allows **one call per day** for backup operation."

Your mind races. Can you ensure that your data is safely preserved multiple times, given this restriction? Can it be done, without asking much of your resources?

This is left as a **bonus challenge**. The solution is yours to discover.