Understanding Microservices: Exploding a Monolith

Nikhil Wanpal

@nikhilwanpal | NikhilWanpal

Today we will

- Part I:
  - Software Evolution in older architectures
  - The philosophy
- Part II:
  - Story of MyKart
  - Case studies
  - The Revolt
- Part III:
  - The Evolution

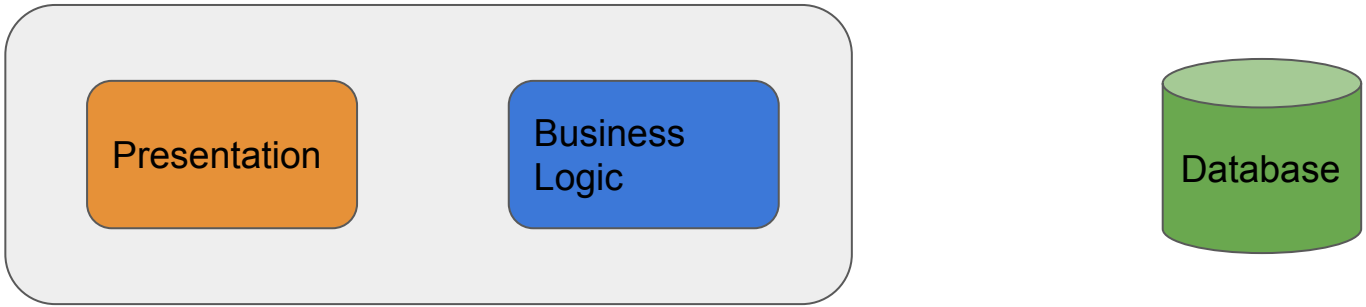
Microservices and Scary Terminologies

- Continuous Integration
- Continuous Delivery
- Continuous Deployment
- DevOps
- NoOps
- SRE
- Infrastructure As Code
- Containerization
- Dockerization
- Container Orchestration Framework
- Private, Public and Hybrid Clouds
- Monolith
- Service Registry
- Service Discovery
- Service Gateway
- Distributed Tracing
- Central Configuration Service
- Log Aggregation Service
- Distributed Monitoring
- Circuit Breakers
- Service Discovery Away REST
- Client Side Load Balancing
- Central Authentication Systems
- Vaults / Secret Managers
- Fault Tolerance
- Self Healing
- Auto Scaling
- Eventual Consistency
- Event Sourcing
- Stateless APIs
- Contract Testing
- Conway's Law
- Agile
- 2 Pizza Teams

Older Problems and our solutions

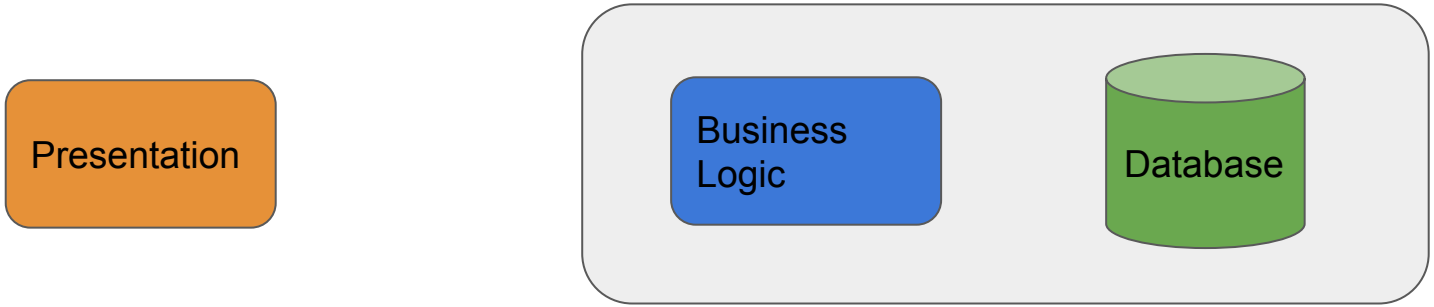
1 Tier application

Older Problems and our solutions



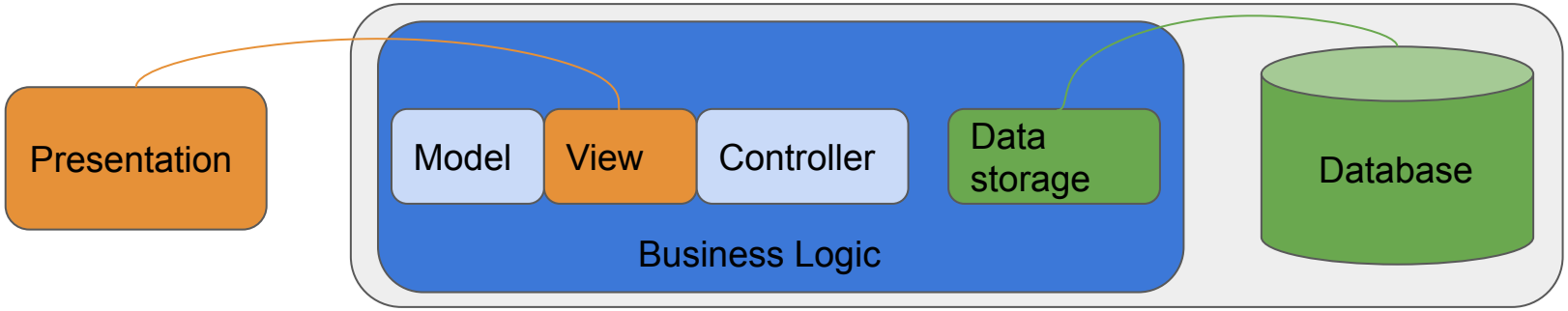
2 Tier application

Older Problems and our solutions



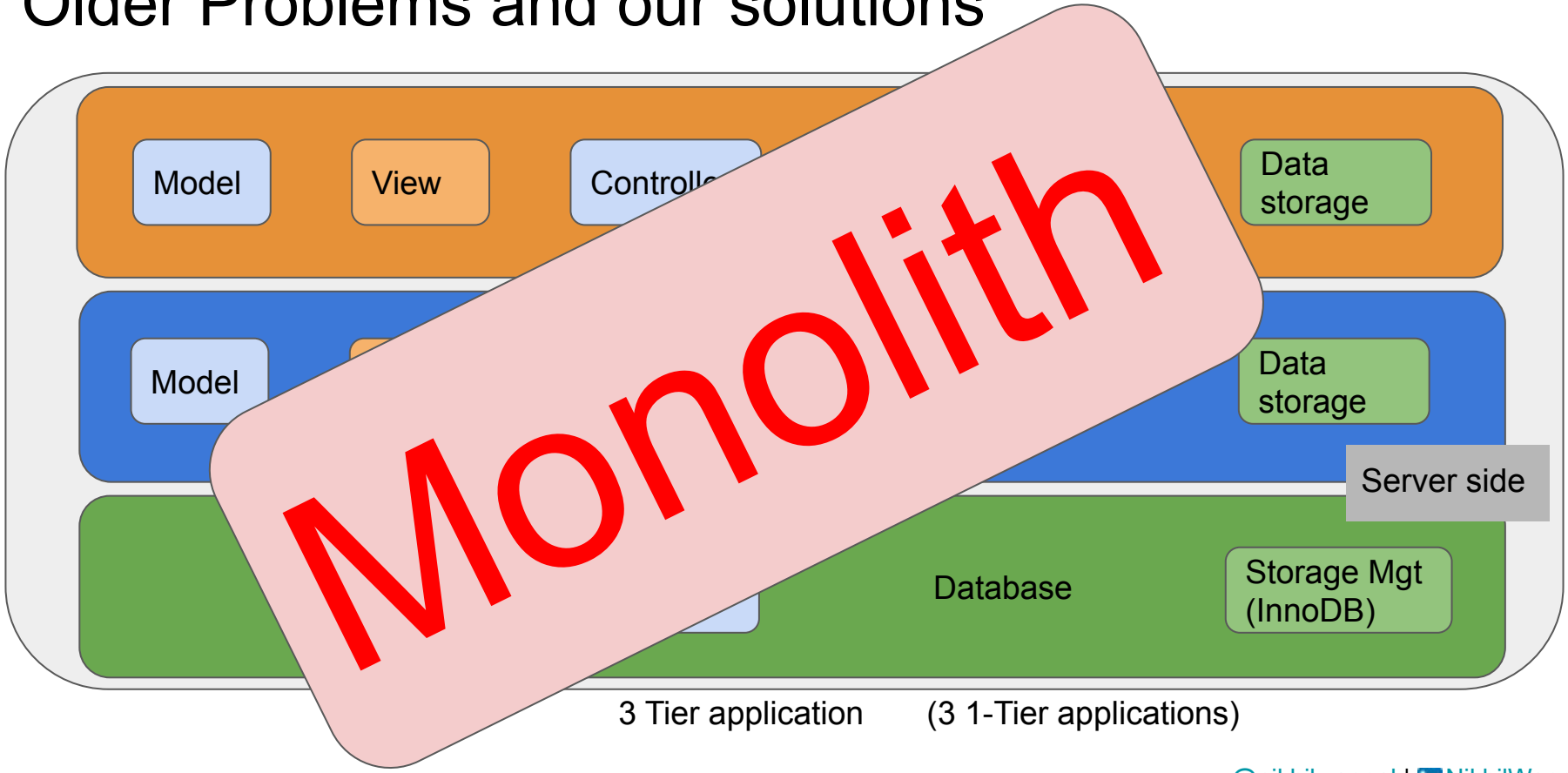
3 Tier application

Older Problems and our solutions

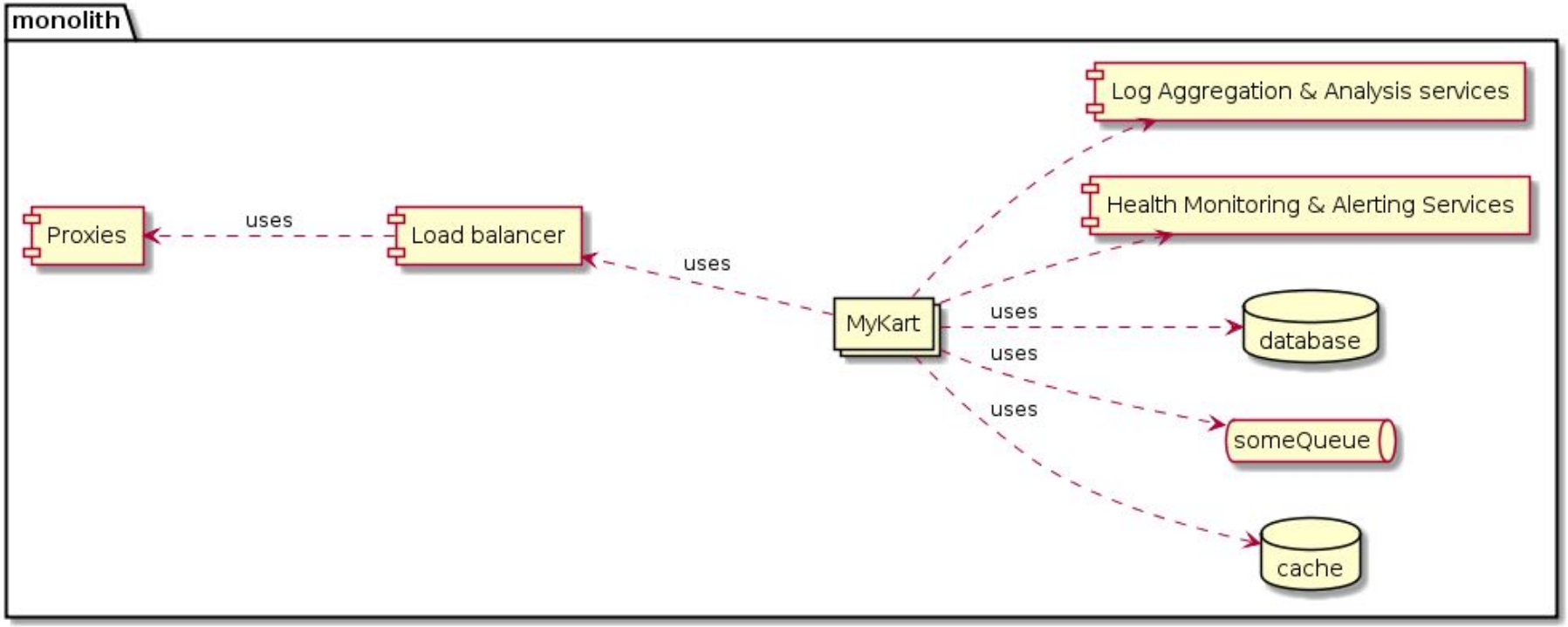


3 Tier application

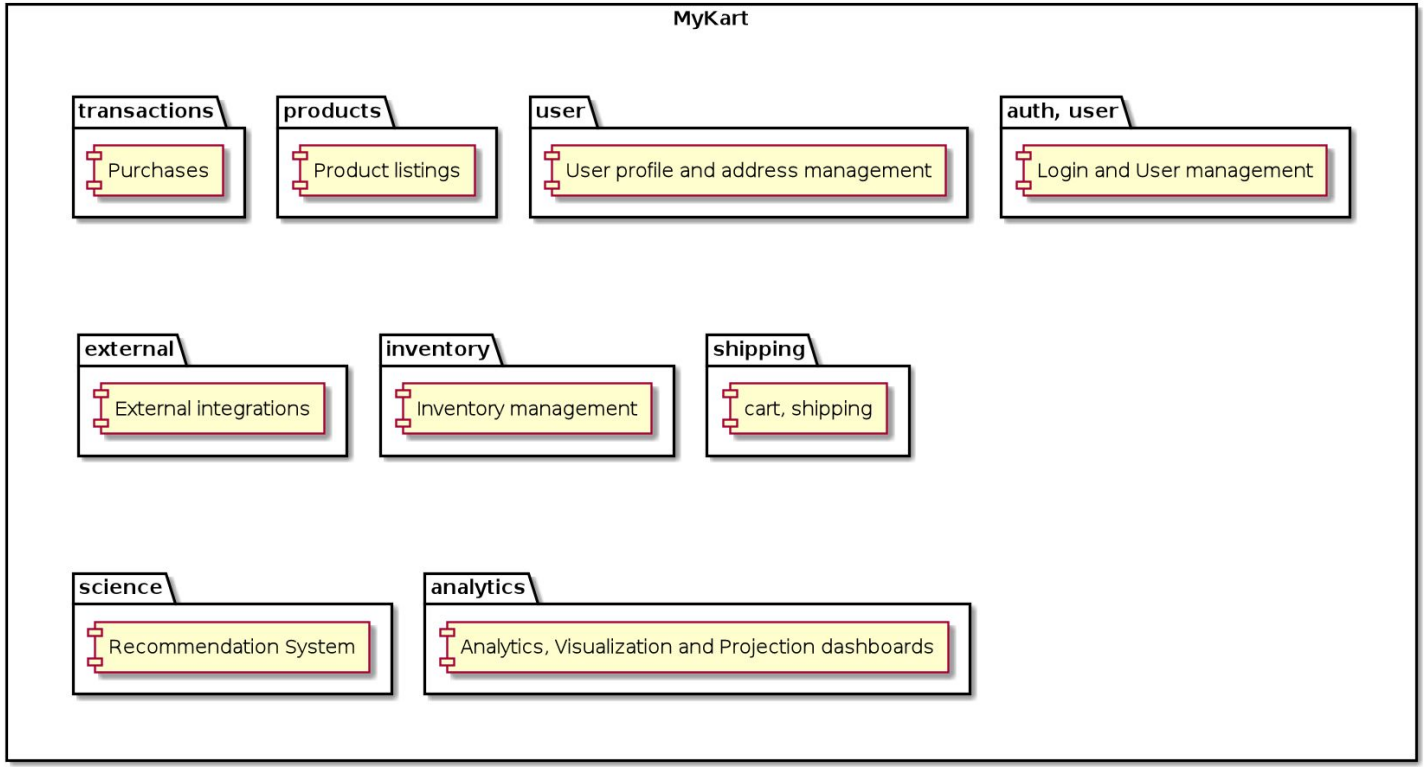
Older Problems and our solutions



A Monolith (In it’s all glory)



A Monolith: The Story of MyKart



What is wrong with a monolith?

- Nothing!
- Perfectly stable, usable, battle tested for applications of all sizes.

- Being monolith:
  - Design & technology applies to all layers
  - Developed as a whole
  - Deployed & scaled as a whole

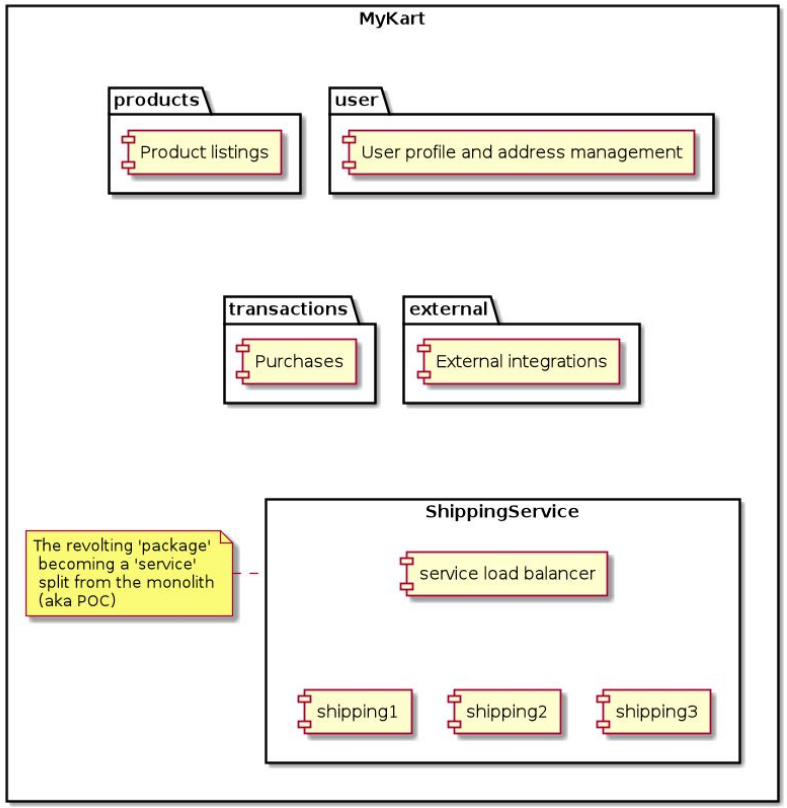
- The nature of monolith is the problem as the application and team grows.

Case Studies

- Load and spikes
- UX Redesign
- New integration on backend
- Framework or library upgrades
- Different teams' (& Business) technology demands
- Business and application size

A revolt!

- I want my team (my architect, UX, devs, testers and DBA) to build our own ‘functionally separate’ part of the system.
- We work at own
  - Pace, Development
  - Technology, Dependencies
  - Release cycle, Deployments
  - Uptime etc.
- We will promise to keep API.



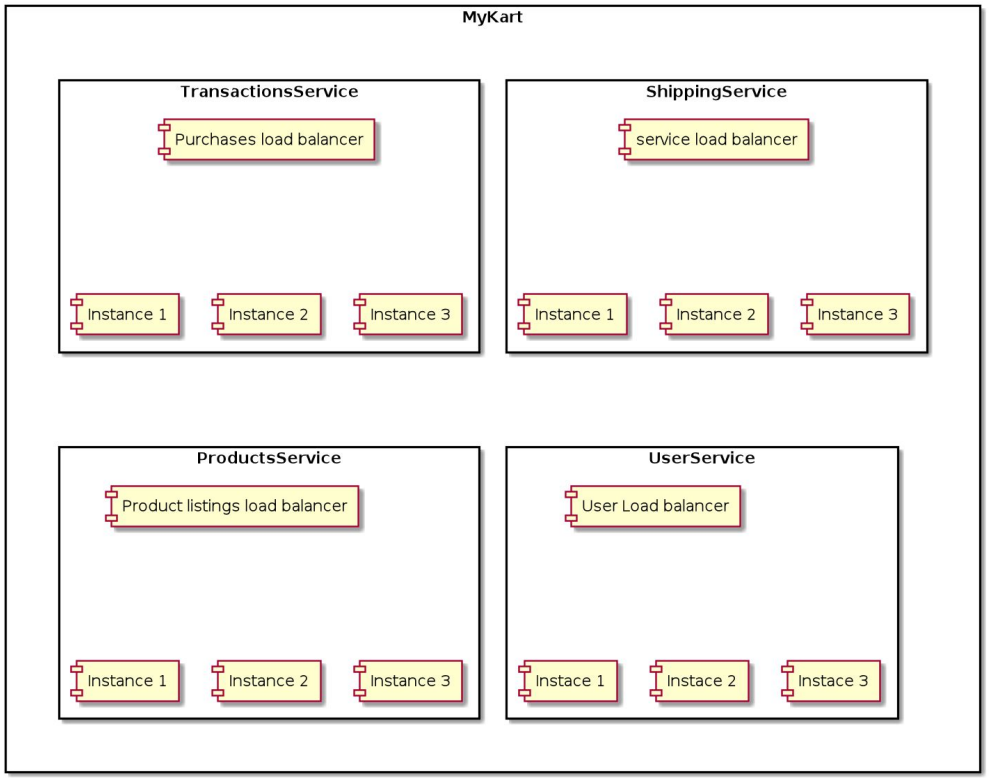
What does that change?

- Technically, not much!
- With preparation and planning, they make do with:
  - *Static IP* and single instance of the Shipping Service load-balancer
  - *Static instances* of shipping service configured on load-balancer
  - Rolling manual / auto deployments to ensure uptime
  - API versioning
- Transaction module -(HTTP)--> ShippingService Load Balancer.
- Intra-process → Inter-process call.
- They make their independent service a partial success!

(deployments are still in same stack and static, scaling is still manual)

We want a taste of it too!

- More services!



What does that change?

- First and foremost: instead of 3, (n * 3 + n) servers/vms!
- Baseline costs go up
- Monitoring and maintenance costs go up
- Inter-process communication issues on rise
- Ops under pressure
- Deployments become difficult, diverse and risky
  - ...and so less frequent, so do releases
  - ...and so does development. (back to square one)

Principles of Software Evolution

(Remember evolution of 1-tier to 3-tier)

- Extract Common: group common, break out different
- Software is all about automation
- We worry less about hardware: developer cycles vs CPU cycles
- Our problems are manifestations of older problems, and so are our solutions
- Simplest solution are best
- We are never satisfied

Solutions

- Too many servers running all the time
  - ✓ Reduce when not needed (aka auto-scaling)
- High instances churn, how do I add them on the load balancer?
  - ✓ automatic registration on load balancers (not discovery yet)
- Load balancer is not enough (auto-scaling and auto registration)
  - Requires a hop
  - Single point of failures in the system
  - Together demand that IPs change, static IPs / DNS won't work anymore.
  - We need a way to map service name to IP.
  - ✓ Hence automatic registration to a central server with service name and instance IP.
  - ✓ Anyone can ask for an address of a service they want! (Aka service discovery)
- Too many servers, they go down all the time
  - Auto restart, ensured minimum instances, Aka self-healing.

Solutions

- Too many servers to deploy
  - ✓ Automate rolling deployments (aka auto deployments, this is not CD ;))
- Too many hardware instances to provision & configure
  - Different technologies, different runtime requirements: configuration and infrastructure
  - ✓ Service specific VM images, infrastructure automation, infra as code (not docker, yet)
- Too many application instances to (re)configure
  - One change to be done in monolith is now ‘n’ changes in ‘n’ places.
  - ✓ External configuration, and a central configuration service and refresh mechanism
- Inter-service communication issues
  - Random failures (apparently), networking issues, load issues, availability issues lead to in general unpredictability.
  - Cascading failures (believe me, these are the worst)
  - ✓ Disable calls when a service misbehaves (aka circuit breaker)

Solutions

- Difficulties in tracing and tracking individual requests
  - Earlier a request could be traced by thread-ids being logged.
  - ✓ Now you need a global, request identification, distributed tracing for better visibility.
- Too many REST calls to make to different IPs, need better clients
  - ✓ ‘Service discovery aware’ rest clients.
- Too many places where retry was required
  - ✓ ‘Service discovery aware’ rest clients aware retriable functionality.
- Too many tools to configure at service boundaries:
  - ✓ Above integrated into circuit breakers.
- Servers replaced, cannot have state.
  - ✓ Externalized sessions and stateless APIs.

Solutions

- With all this central load balancers make less and less sense
  - Caller:
    - Knows where the service and discovery server is
    - Knows Discovery knows which instance is closer
    - Decides the handling on failure, fallbacks, caches etc.
  - Central load balancer now seems like an unnecessary hop and moving part.
  - ✓ Client side load balancing.
- Too many places to monitor for failures
  - ✓ Central health check and service monitoring systems.
- It is impossible to find the instance on which the error occurred.
  - ✓ Central log repositories and searches.

Solutions

- Too many places to authenticate and authorize users at (when multiple entry points)
  - ✓ Central authentication systems
- Too many places to authenticate and authorize services at
  - ✓ Secrets management systems
- Does our data center machines have this
  - ○ Fancy ability to spin up and shut down, this scaling thing?
  - ○ Can we incorporate easily with all the networking things we need?
  - ○ Ohh it is just too many servers to look after
  - ✓ Private / public clouds and managed services

Solutions

- But then what do we do with all the money we put in buying this data center?
  - ✓ We can use it, Hybrid cloud setups!

Are things getting clearer now? Well this isn’t over yet!

Solutions, allied problems

- We are faster
  - there is just too much to test
  - we need ‘moar’ speed in validation and packaging.
  - ✓ Automated testing and Continuous Integration (CI)
- Can we move even faster now that we have:
  - Automation everywhere
  - High uptime ensured by self-healing services
  - Easy reversals enabled through infra and deployment versioning
  - ✓ Yes, Continuous Delivery (CD)
- More speed? Now that we are confident it works?
  - ✓ Continuous Deployment, if ecosystem favours.

Solutions, allied problems

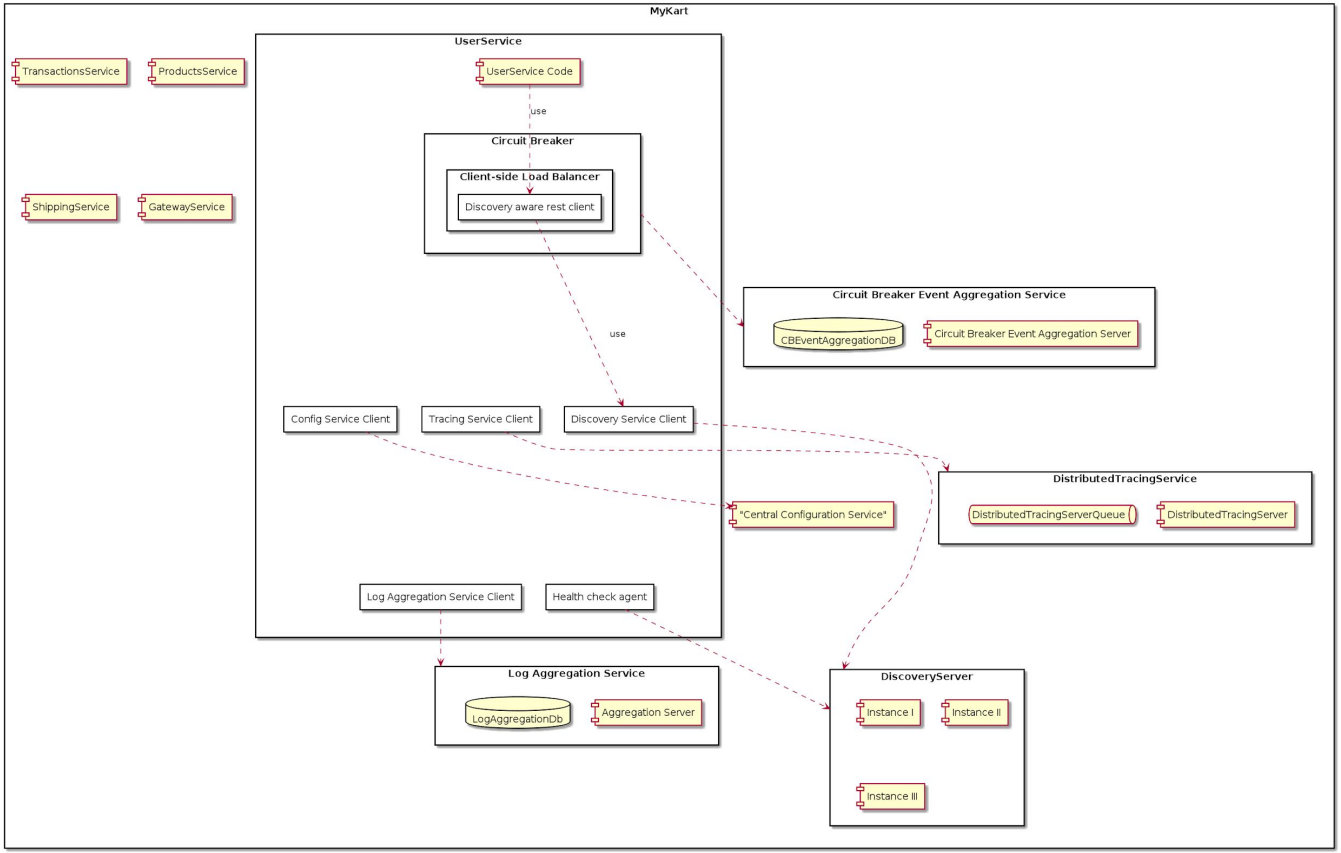
- Costs are still high
  - Many VMs are still live
  - Not all fully utilize their RAM/CPU at idle

- Development difficulties
  - Too much to set up locally
  - Dev machines loaded
  - So many dependencies to manage
  - How to ensure consistency in environments?

Solutions, allied problems

- Deployment difficulties. Each service has:
  - Specific infra requirements
  - Own specific deployment steps and dependencies
  - Central SRE / Ops overloaded, changes pushed back.
  - Devs feel limited in not having the control over infra (DevOps anyone?)
- For all 3 above:
  - ✓ Containerize the stack
  - Let developers create the ‘VM’ images, manage what goes in it
  - There is a consistent API to start and stop any service
  - So, it is standardized now?! Well, then why do we need OPs, infra? Use:
  - ✓ Container Orchestration Frameworks, like Kubernetes

The Monolith becomes



Are we there yet?

Solutions and allied problems

- Inter team clashes
- Push backs to changes
- Difference in priorities and goals of teams
- Scaling limitations

- Conway’s law: Software architecture reflects the organization’s communication structure.
- Organizational restructuring

Solutions and allied problems

- Agility
- Unit Tests, Integration Tests, System Tests, Contract Tests
- Automated Testing End-To-End testing
- CI
- CD
- DevOps
- NoOps

- We need these all!

Are we there yet?

Probably..

Before Microservices, Ask Yourself

- Why?
- Do you understand your business good enough to be able to clearly segregate your application components into individual services? (Functional segregation)
- Is the application large enough to require micro services?
- Is the team large enough to own up individual components?
- Is the complexity worth the benefits you seek?
- Is your organization ready to support your architecture?
- Anything else you would like to add?

Disclaimers

- Some things were generalised and (over)simplified
- Retrospectively, things always seem obvious and natural
- Some stories were made up

```
}
```

Todos to self-discovery:

Event Sourcing.

Eventual Consistency and microservices.