High Availability

Renato Panda

renato.panda@ipt.pt

Back to ACME Inc…

- David’s life is going great now at ACME
  - Virtualization is being widely used
    - Easier backups and migrations
    - Reduced server sprawl
    - Reduced costs
  - DevOps culture was implemented
    - Measures to automate testing, integration, deployment, (…)

Back to ACME Inc…

- David’s life is going great now at ACME
  - Load balancing is also being used
    - Provides redundancy for essential services
    - Higher loads handled by N nodes

Back to ACME Inc...

- Still, David identifies new problems
  - Database layer
    - What if the DB goes down?
    - Can I load balance in a relational DB?
  - Load balancing
    - What if the load balancer crashes?
    - Can I load balance the load balancer?

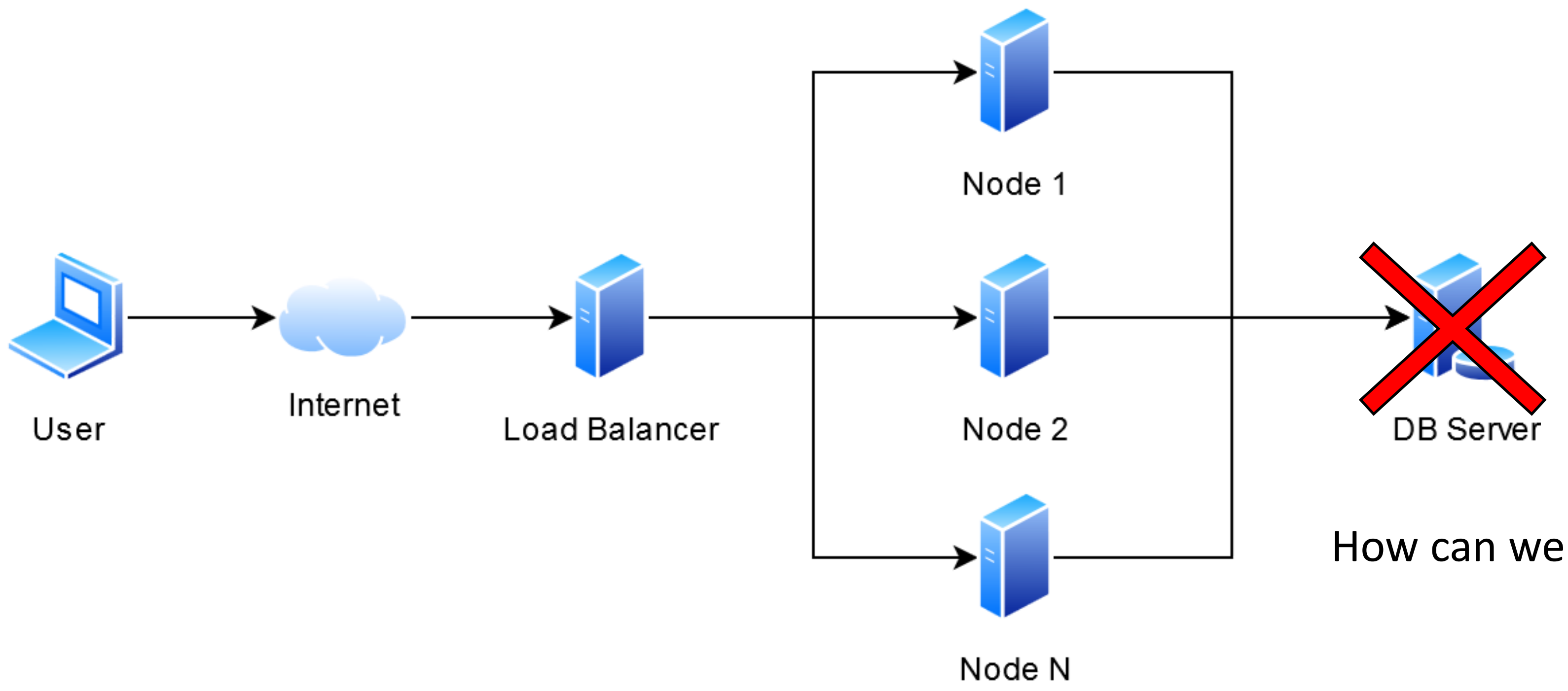
High Availability (HA)

- What is an HA system?
  - System that ensures an agreed level of operational performance, usually uptime, for a higher-than-normal period

  - How high is high enough?
    - 90% = 36.53 days/year (downtime)
    - 99% = 3.65 days/year
    - 99.9% = 8.77 hours/year
    - 99.99% = 52.60 minutes/year
    - 99.999% = 5.26 minutes/year
    - 99.9999% = 31.56 seconds/year

Problem #1: Database Layer

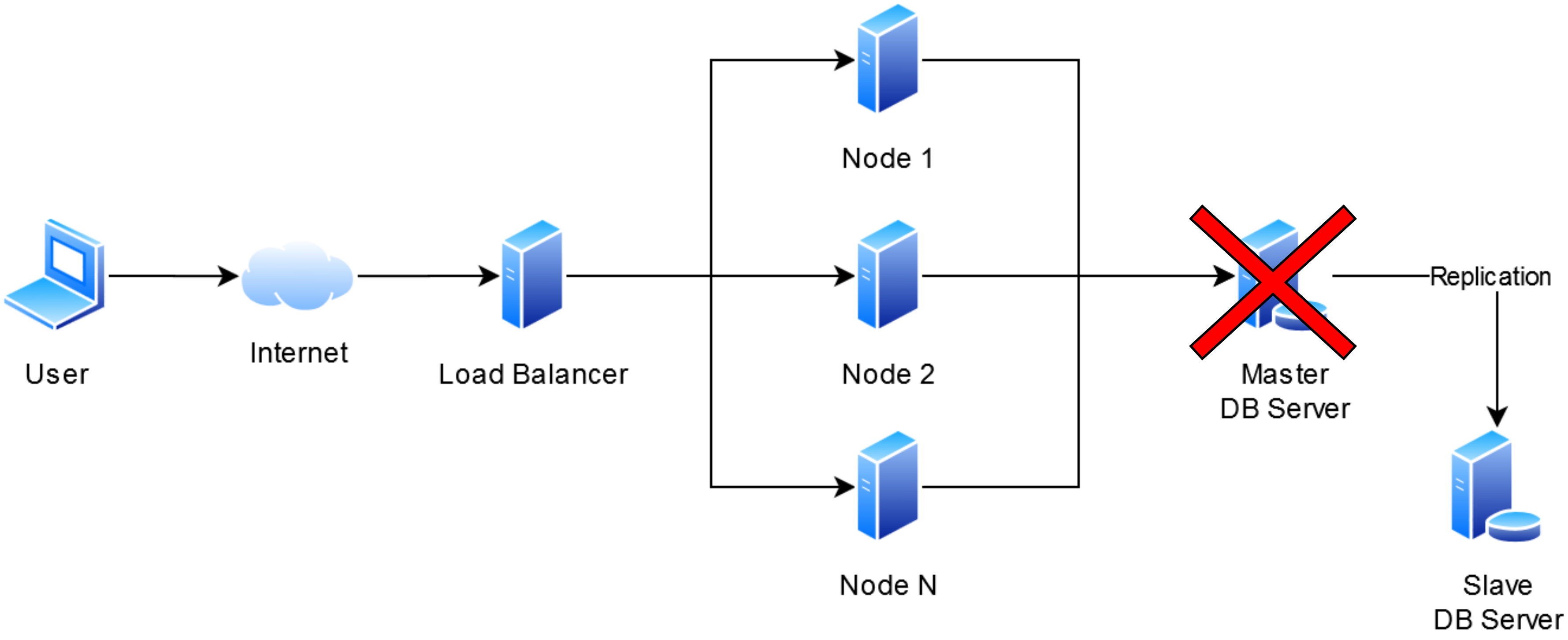
- What if the database goes down?
  - Single DB Server? Application layer cannot get data (**single point of failure**)



How can we improve this?

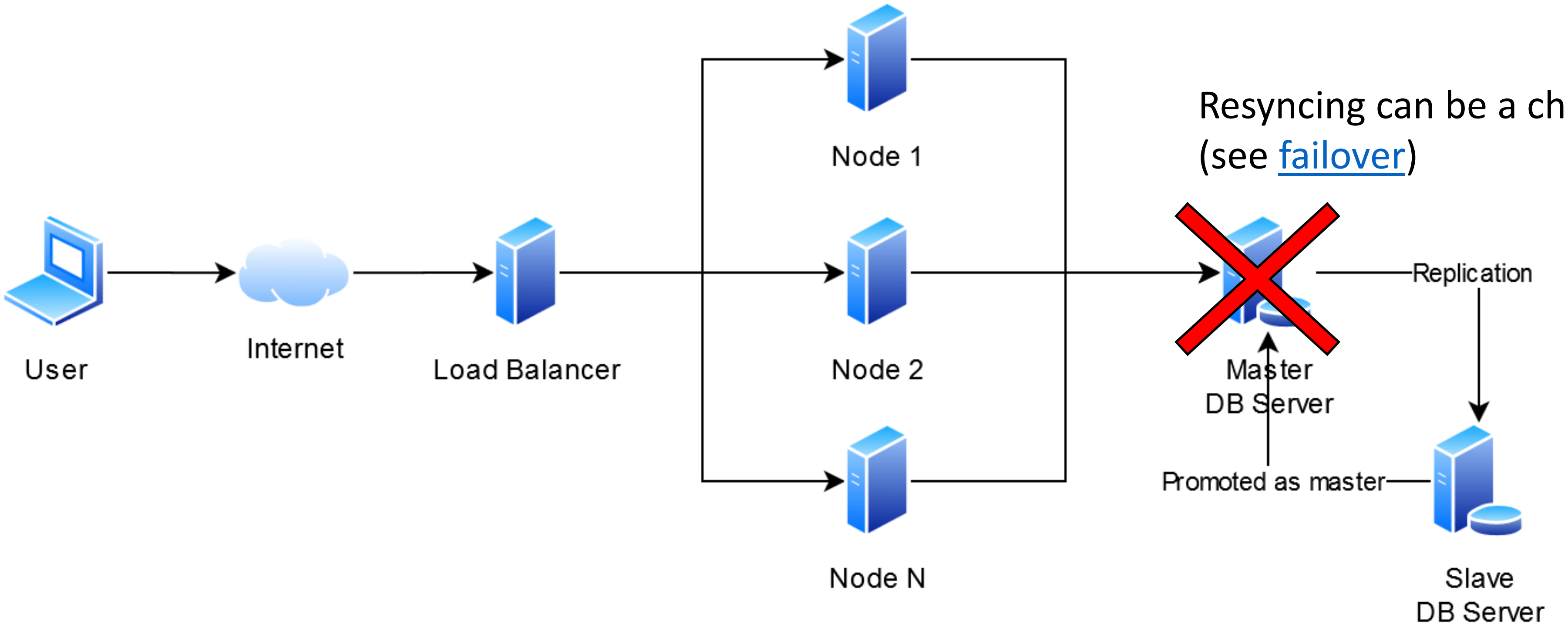
Problem #1: Database Layer

- What if the database goes down?
  - Master / slave configuration (data is replicated to the second server)



Problem #1: Database Layer

- What if the database goes down?
  - Master / slave configuration (if master fails, slave is promoted to master)

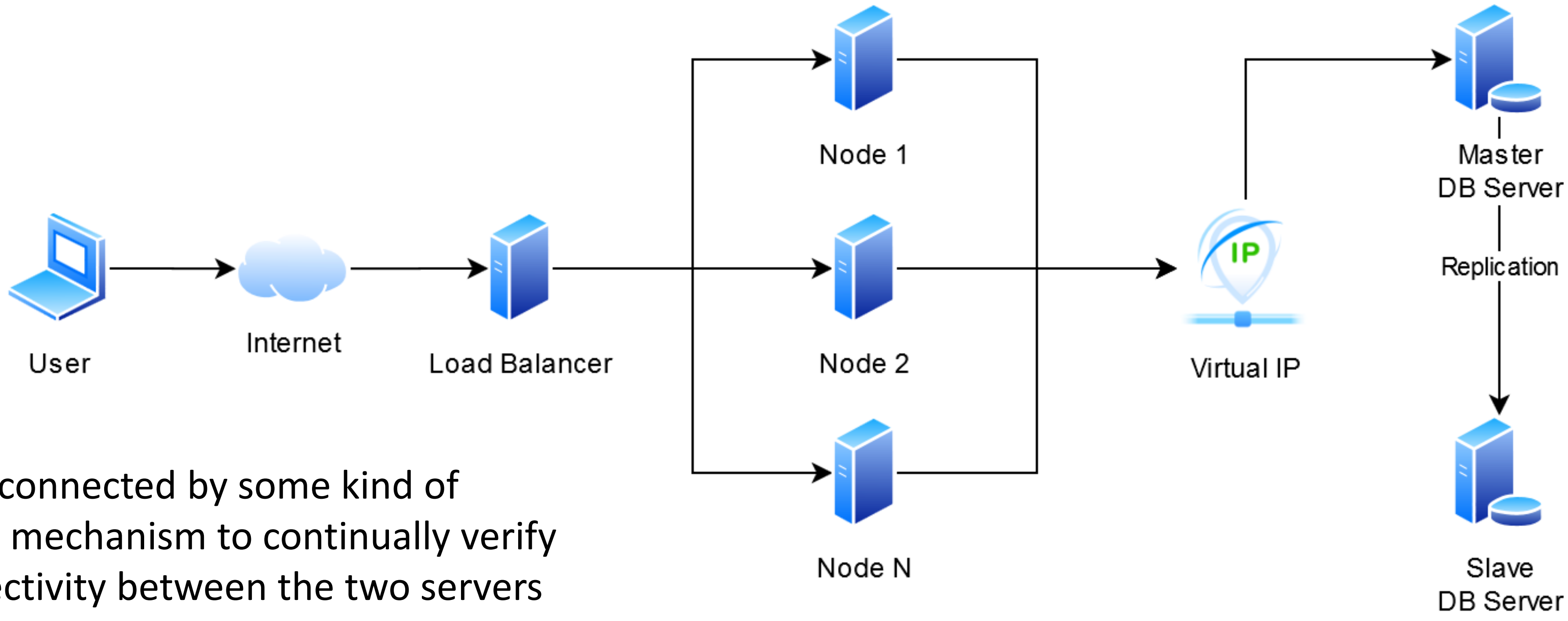


Resyncing can be a challenge (see failover)

Only a single server in operation is known as "degenerate state"

Problem #1: Database Layer

- What if the database goes down?
  - Master / slave configuration (resync data can take time – e.g., *pg_rewind* tool)



Normally connected by some kind of heartbeat mechanism to continually verify the connectivity between the two servers

Problem #1: Database Layer

- Several solutions for replication in DBMS (e.g. PostgreSQL)
  - Shared Disk Failover
  - File System (Block Device) Replication
  - Write-Ahead Log Shipping
  - Synchronous and Asynchronous Replication
  - Several others

Problem #1: Database Layer

- 1st step: Database Replication

- What?
  - Keeping multiple copies of the DB

- Why?
  - Availability
    - What if the DB goes down?
  - Latency
    - What if accesses are from around the world?
  - Scalability (more about this later)
    - What if we have millions of accesses?

Problem #1: Database Layer

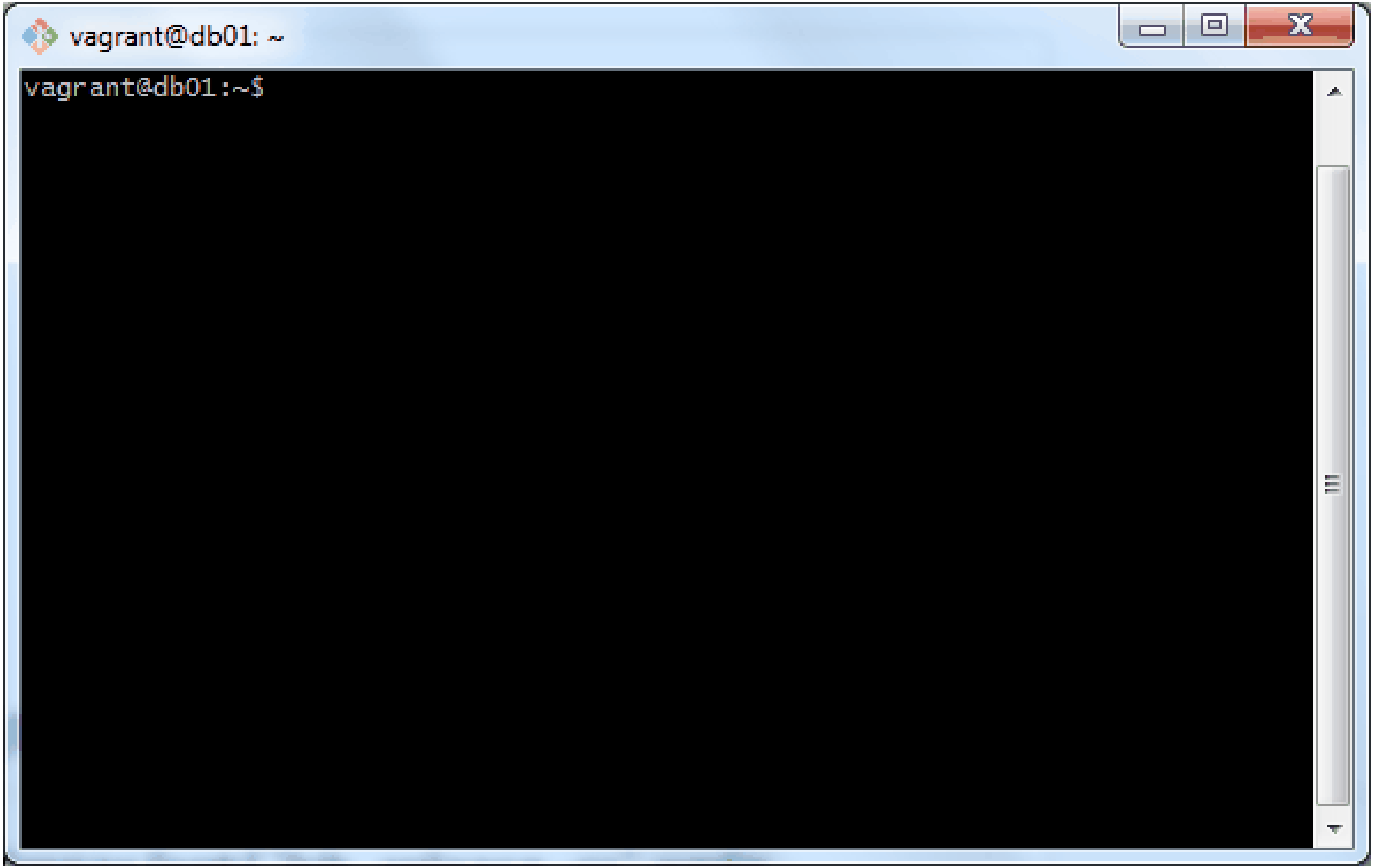
- Configuring replication in PostgreSQL
  - Requirements
    - Master – 192.168.33.50
    - Slave – 192.168.33.60
  - Challenge
    - Provision PostgreSQL via ansible

```
Vagrant.configure("2") do |config|
  config.vm.define "db01" do |db01|
    db01.vm.box = "bento/ubuntu-16.04"
    db01.vm.hostname = "db01"
    db01.vm.network :private_network, ip: "192.168.33.50"
    db01.vm.provider "virtualbox" do |v|
      v.memory = 512
    end
    db01.vm.provision "shell", path: "install_postgresql.sh"
  end

  config.vm.define "db02" do |db02|
    db02.vm.box = "bento/ubuntu-16.04"
    db02.vm.hostname = "db02"
    db02.vm.network :private_network, ip: "192.168.33.60"
    db02.vm.provider "virtualbox" do |v|
      v.memory = 512
    end
    db02.vm.provision "shell", path: "install_postgresql.sh"
  end
end
```

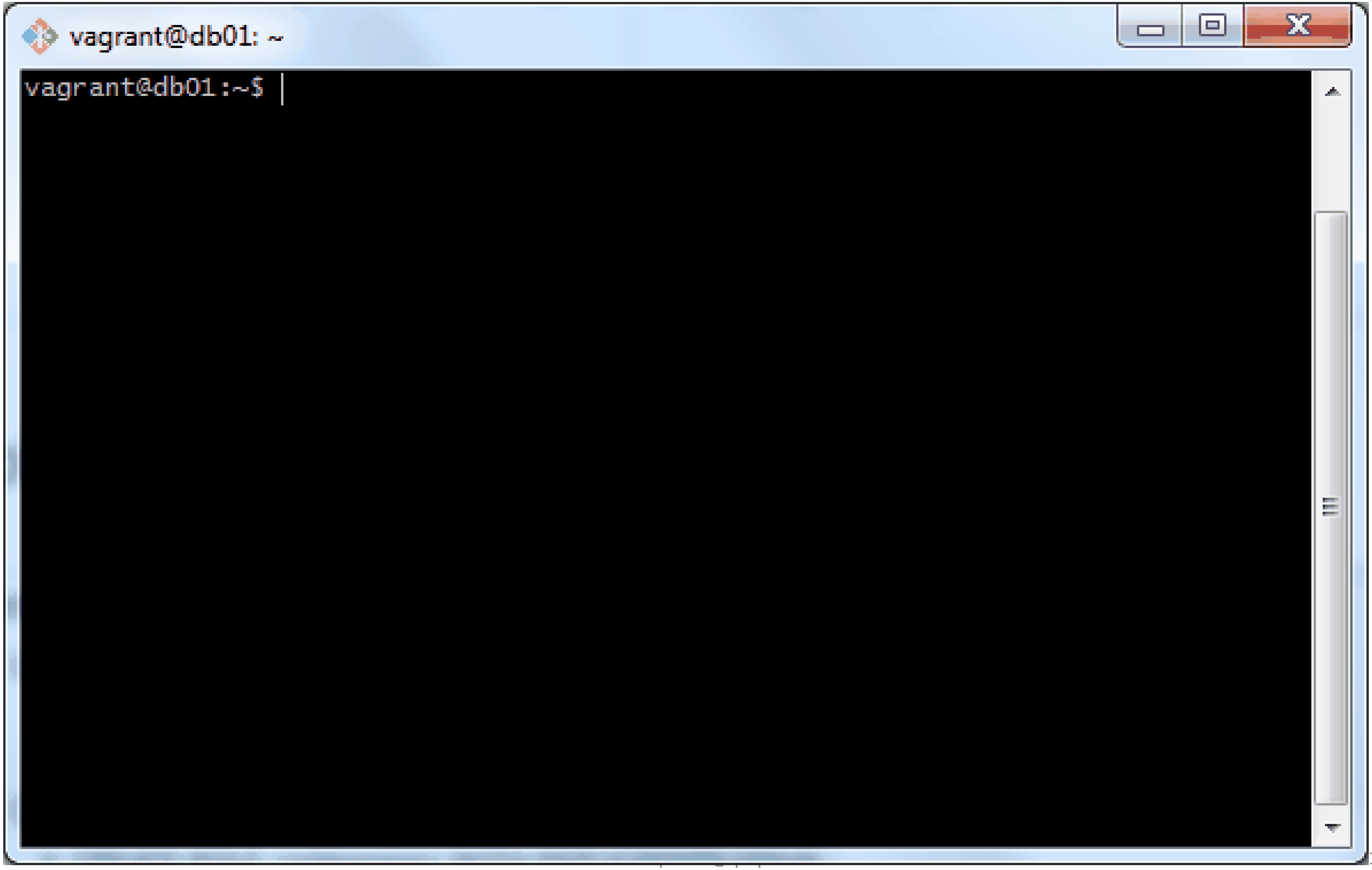
Problem #1: Database Layer

- Installing PostgreSQL (follow updated docs?)
  - Step 1 – Enable PostgreSQL Apt Repository
    - `sudo apt-get install wget ca-certificates`
    - `wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -`
    - `sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt/ `lsb_release -cs`-pgdg main" >> /etc/apt/sources.list.d/pgdg.list'`
  - Step 2 – Install PostgreSQL on Ubuntu
    - `sudo apt-get update -y`
    - `sudo apt-get install postgresql postgresql-contrib -y`


vagrant@db01: ~
vagrant@db01:~$

Problem #1: Database Layer

- Configure Master Node
  - Login to PSQL (default user = postgres)
    - `su - postgres`
    - `psql`
  - Create a new ROLE used for replication
    - `CREATE ROLE <rolename> WITH REPLICATION LOGIN ;`
  - Change password encryption
    - `set password_encryption = ‘scram-sha-256’;`
  - Set the password:
    - `\password <rolename>`

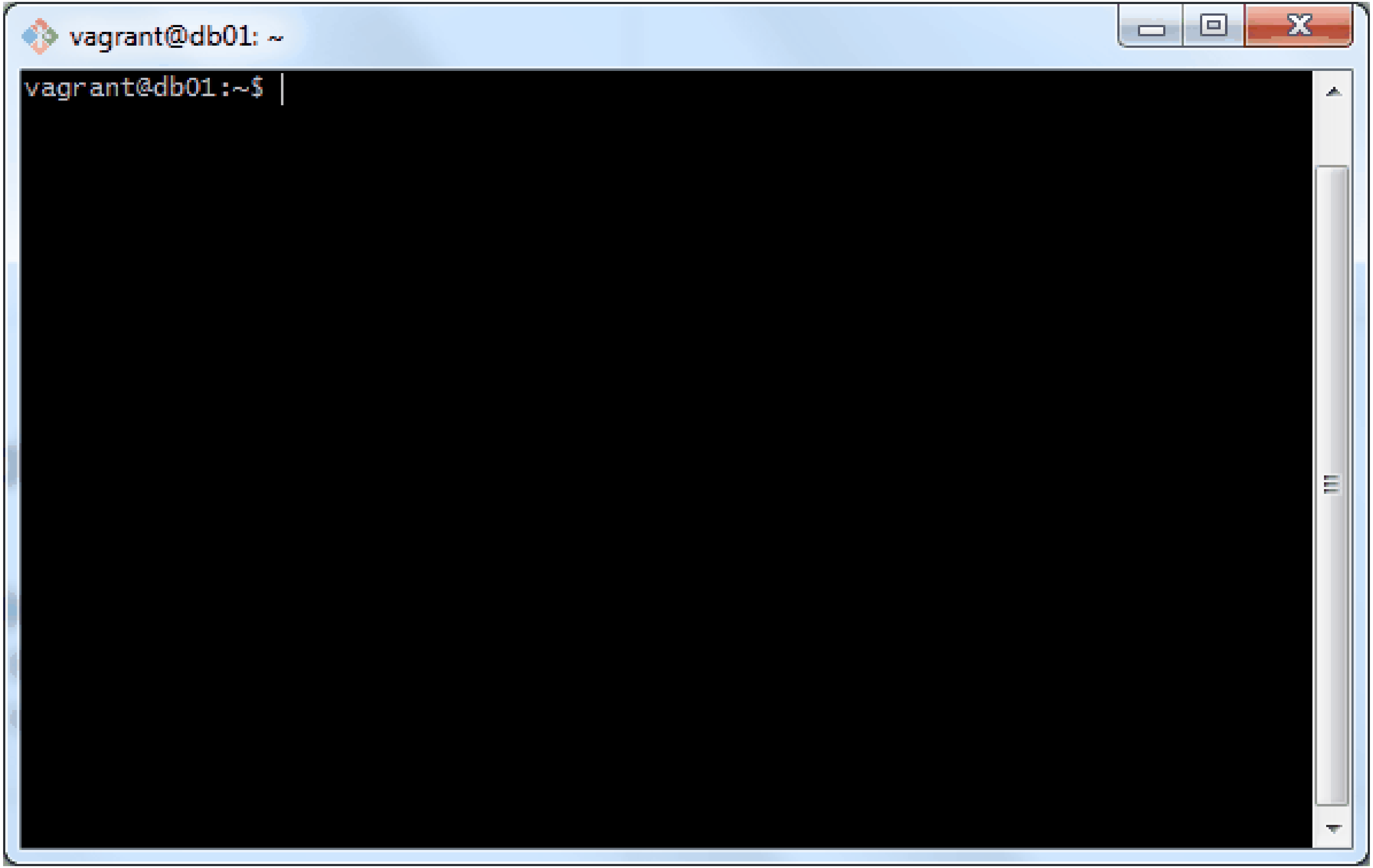

vagrant@db01: ~
vagrant@db01:~$

Problem #1: Database Layer

- Adjust Master for replication (stop the service first)
  - Edit /etc/postgresql/12/main/postgresql.conf (part 1):
  - Setup listen on specific ***if***
    - `listen_addresses = ‘*’ (or the ethX addr)`
  - Adjust WAL settings:
    - `wal_level = replica`
    - `archive_mode = on`
    - `max_wal_senders = 3`
    - `wal_keep_segments = 64 (each segment is 16 MB)`

Problem #1: Database Layer

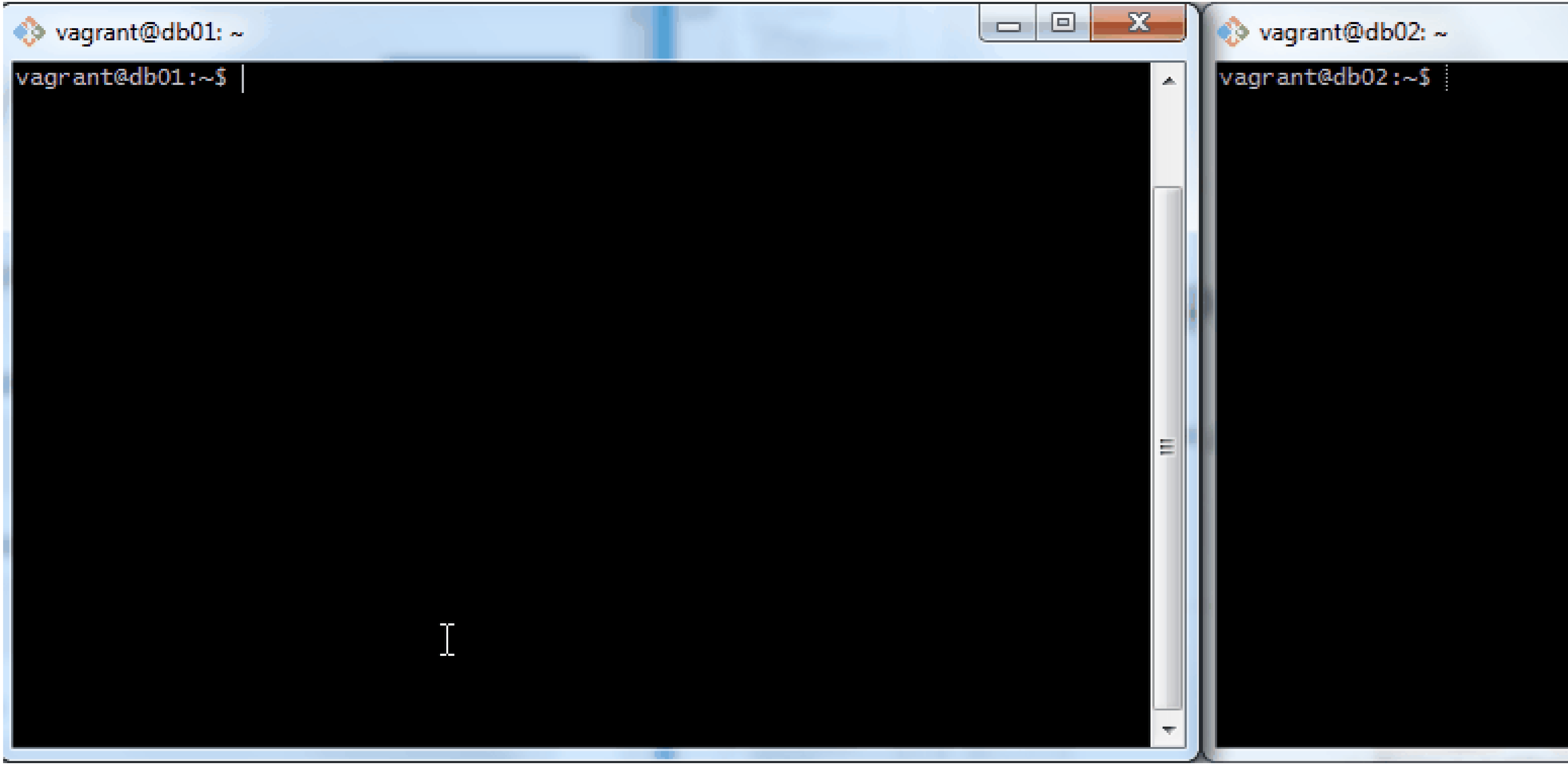
- Adjust Master for replication
  - Edit /etc/postgresql/12/main/postgresql.conf (part 2):
  - Use *rsync* to archive the logs to a specific location
    - archive_command = ‘rsync -a %p postgres@<slaveHost>:/var/lib/postgresql/12/main/archive/%f’
  - Notes:
    - rsync needs to be able to SSH into the host
    - “archive” folder needs to be created:
      - mkdir -p /var/lib/postgresql/12/main/archive/
      - chmod 700 /var/lib/postgresql/12/main/archive/
      - chown -R postgres:postgres /var/lib/postgresql/12/main/archive/


vagrant@db01: ~
vagrant@db01:~$

Problem #1: Database Layer

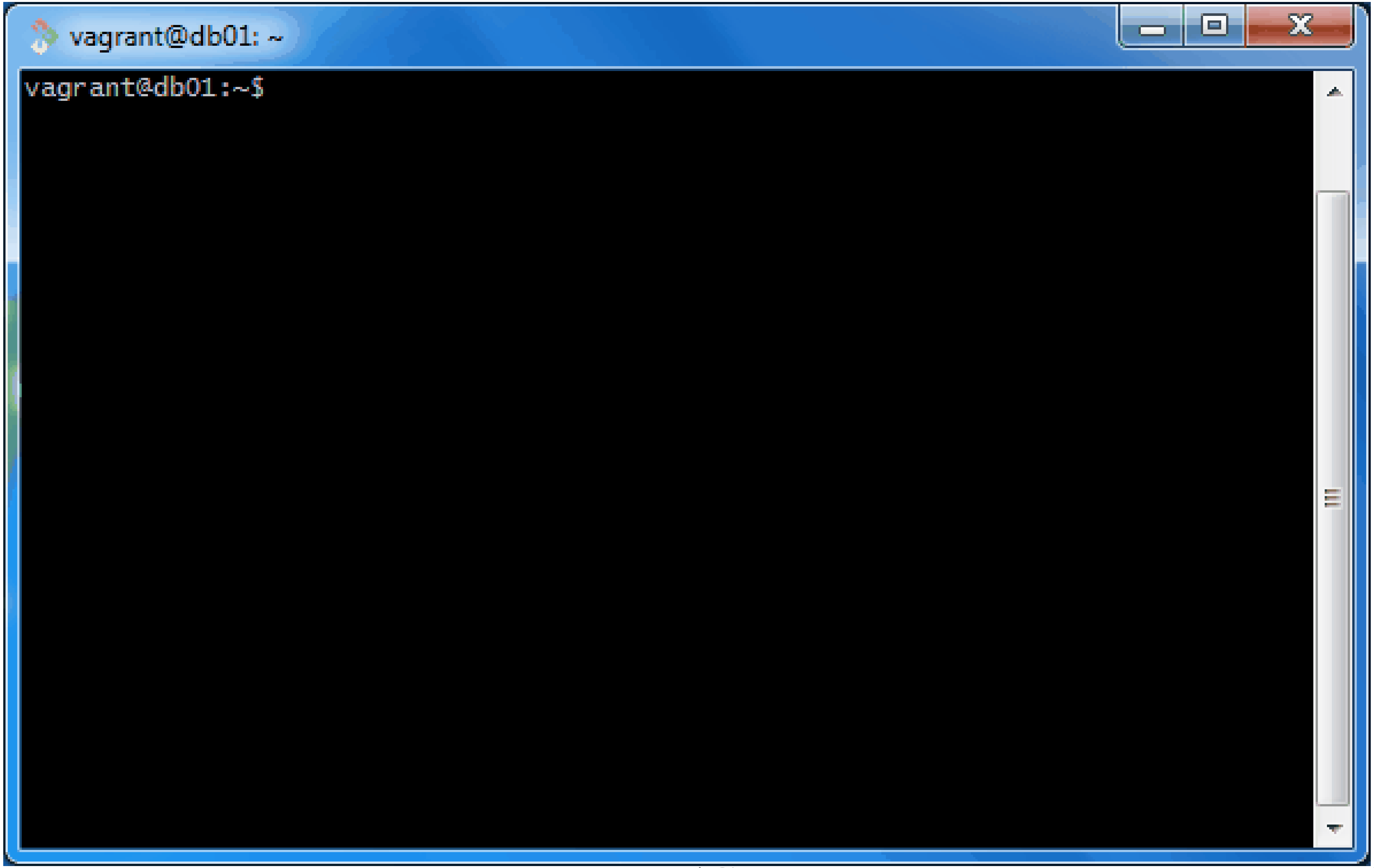
- Adjust Master for replication
  - archive_command will copy the WAL logs to the replica / slave using rsync
    - Different solutions can be used. E.g., what if I want several replicas?

    - In our scenario, rsync needs to be able to SSH into the host (passwordless!)
      - Test *rsyncing* to db02 as postgres
      - Setup pwd login for postgres@db02
      - ssh-keygen && ssh-copy-id from db01 to db02
      - Remove pwd login for postgres@db02
      - Test rsync again


vagrant@db01: ~
vagrant@db01:~$
vagrant@db02: ~
vagrant@db02:~$

Problem #1: Database Layer

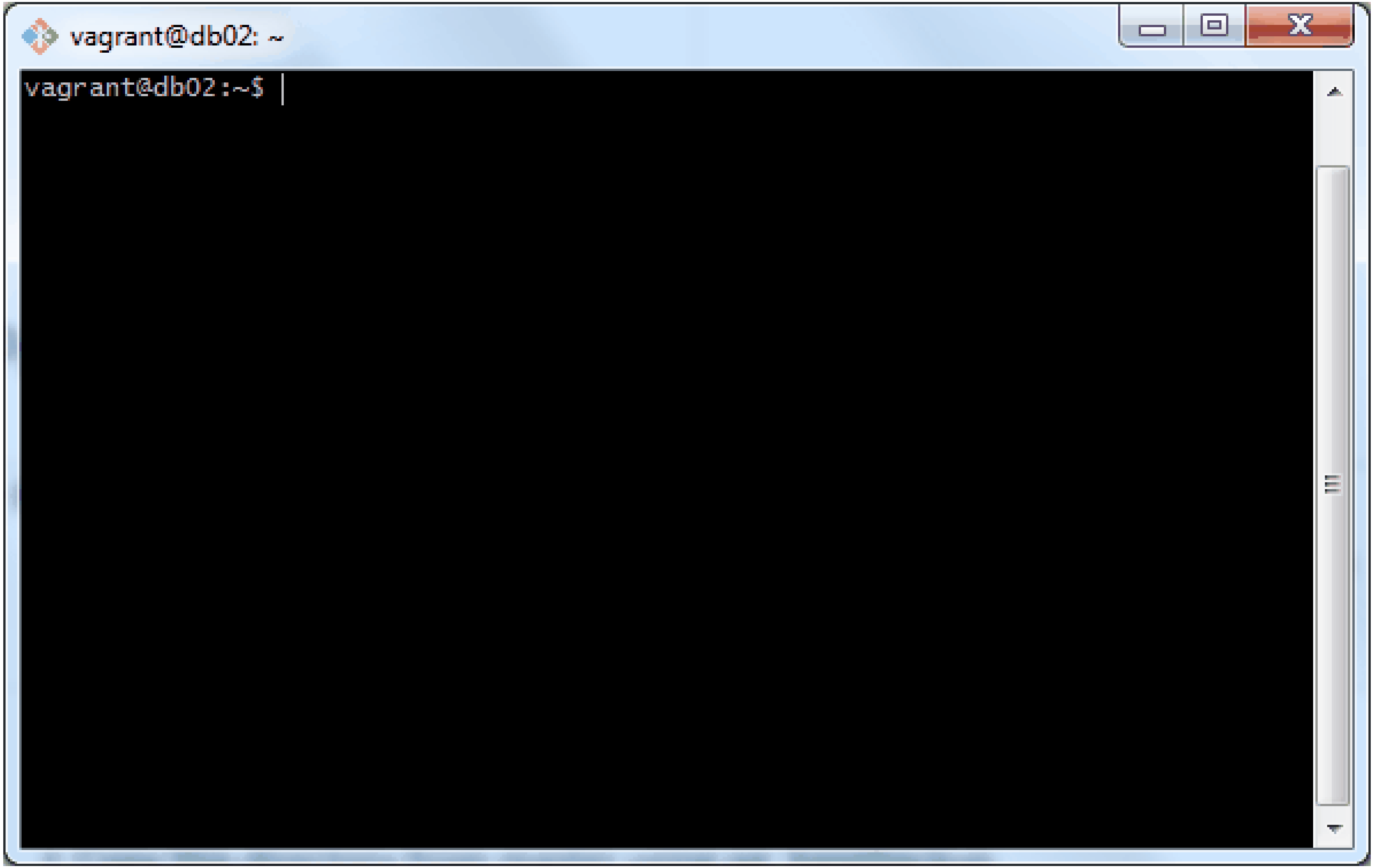
- Allow the Slave Host to connect to master
  - Edit /etc/postgresql/12/main/pg_hba.conf (@ db01) to allow the slave IP address (what if we had several replicas?)
  - Add to the end of the file:
    - “`hostssl    replication    <rolename>     <ip/network>          scram-sha-256`”


vagrant@db01: ~
vagrant@db01:~$

Problem #1: Database Layer

- PostgreSQL Slave Configuration
  - Stop the server
    - `systemctl stop postgresql`
  - Activate hot_standby in postgresql.conf:
    - `hot_standby = on`
  - Create a new pgdata directory
    - Move/backup/create the directory `/var/lib/postgresql/12/main`
  - Copy this directory from master using pg_basebackup
    - `pg_basebackup -h <master> -D /var/lib/postgresql/12/main/ -P -U <role> --wal-method=stream`

Hot Standby? Warm Standby?


vagrant@db02: ~
vagrant@db02:~$

Problem #1: Database Layer

- PostgreSQL Slave Configuration [recovery!]
  - Before v12, the presence of main/recovery.conf file triggered the server into recovery upon start
  - In addition, the file contained parameters to do the recovery, e.g.:
    - `standby_mode = 'on'`
    - `primary_conninfo = 'host=<master> port=5432 user=<role> password=<pwd>'`
    - `trigger_file = '/tmp/MasterNow‘ # slave steps in as master if this file exists`
    - `#restore_command = command to restore archived WAL segments, e.g.:`
    - `restore_command = 'cp /var/lib/postgresql/12/main/archive/%f %p'`

  - After recovery, the “recovery.conf” file was renamed to “recovery.done”

Problem #1: Database Layer

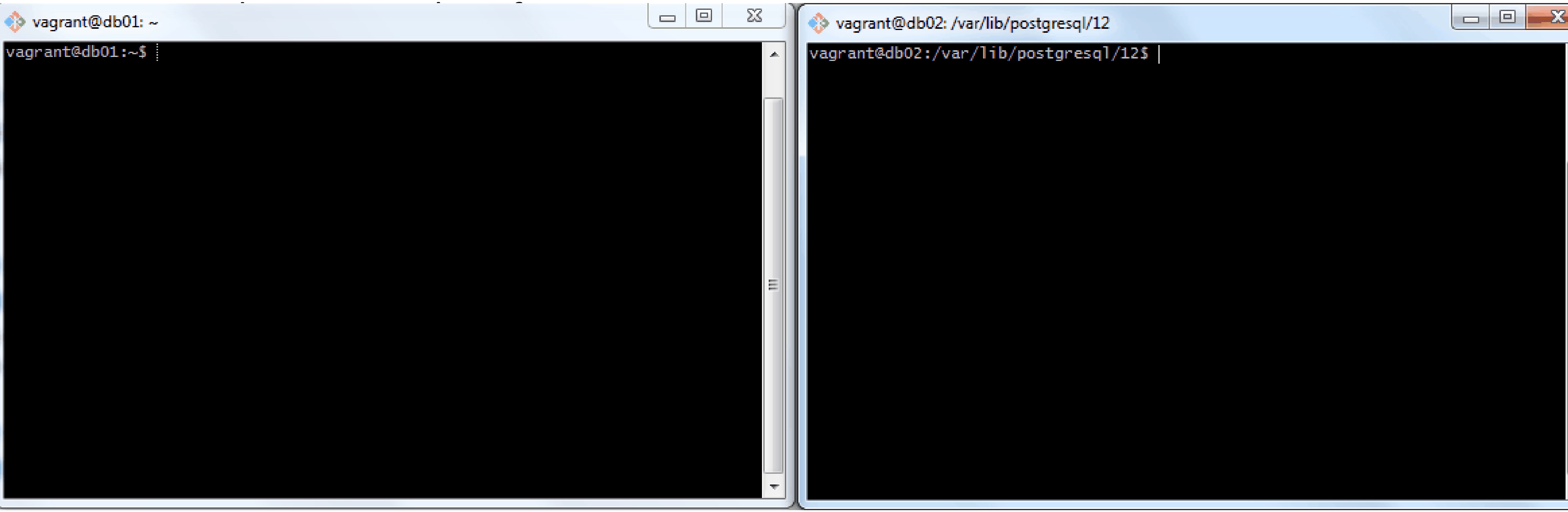
- PostgreSQL Slave Configuration [recovery!]
  - Since v12, support for “recovery.conf” was removed, all parameters are stored in posgresql.conf (avoids having settings in several different files)
    - Also, the “standby_mode” parameter has been removed
  - Two new files exist now:
    - recovery.signal: tells the server to enter normal archive recovery
    - standby.signal: tells the server to enter standby mode

Problem #1: Database Layer

- PostgreSQL Slave Configuration [we need to standby, not to recover]
  - Edit postgresql.conf
    - Add primary_conninfo
    - Add restore_command
    - Add recovery_target_timeline = 'latest‘ (missing in the demo of the next slide)
    - Touch standby.signal

Problem #1: Database Layer

- PostgreSQL Slave Configuration [we need to standby, not to recover]



Some of the previous steps are repeated in this demo

Problem #1: Database Layer

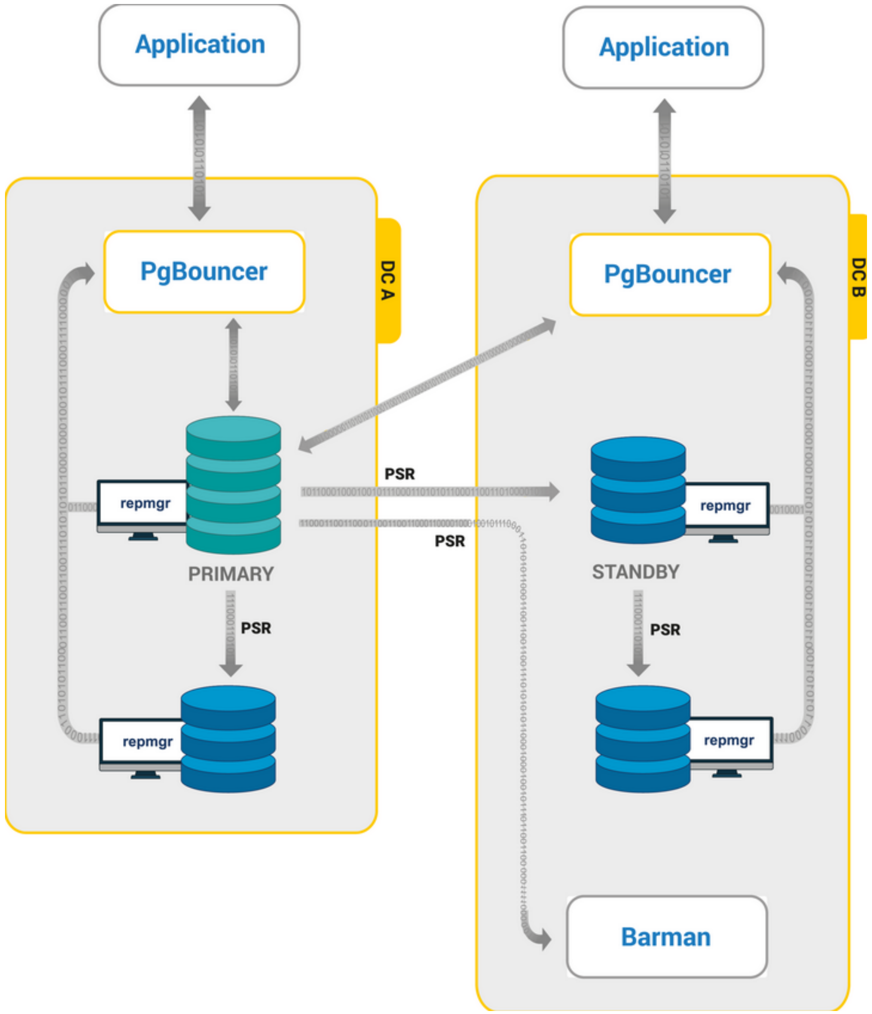
- What did we achieve?
  - PostgreSQL master / replica setup
    - Replica in hot-standby (responds to read-only queries)
      - Can be used to balance load (easy to have multiple replicas / read-only cluster)

    - What about failover?
      - The replica should be able to become master
      - What if the master gets back / cannot contact replica?
        - Split-brain situation
        - Important to STONITH ("Shoot The Offending Node In The Head“)
      - What about the applications using the current master?

Problem #1: Database Layer

- Several dedicated software tools exist
  - Backups
    - [pgBackupRest]
    - [pg_probackup]
    - [Barman]
  - Replication
    - [repmgr]
  - Connection Pooler
    - [PgBouncer]
    - [pgpool]

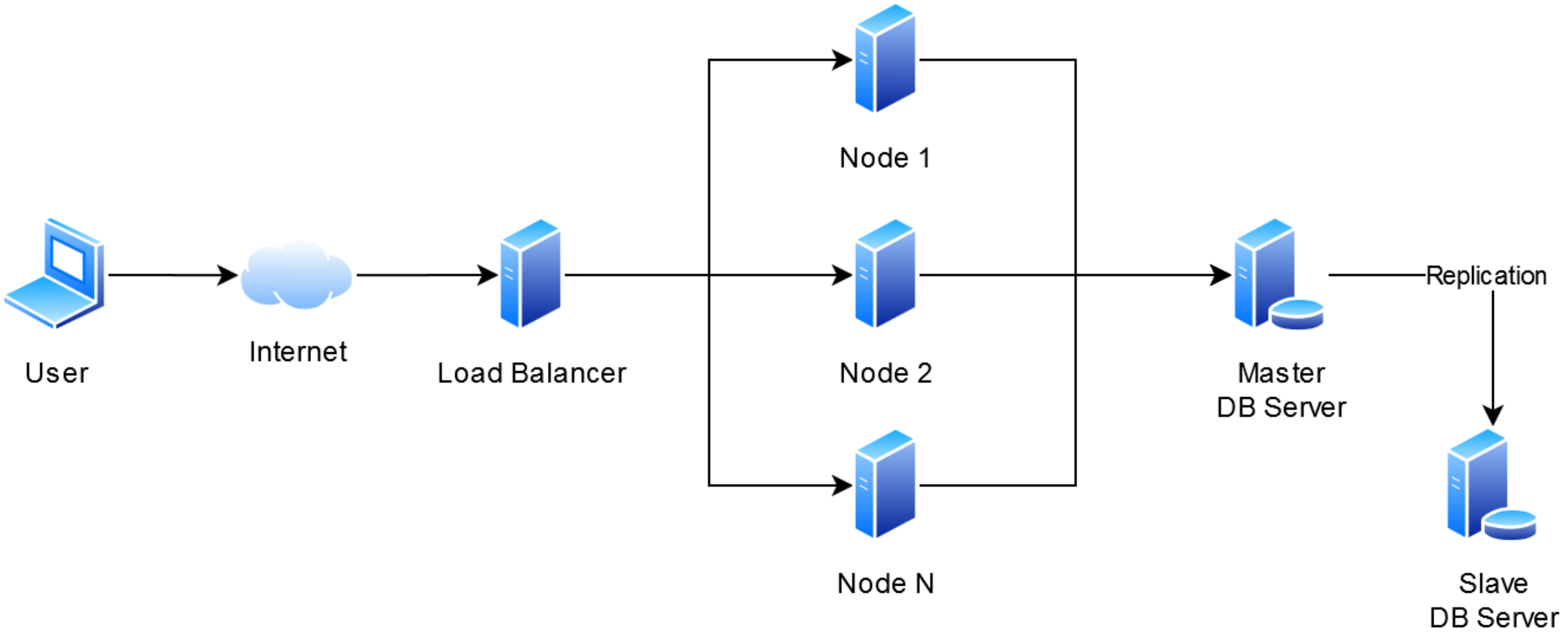
This is just for PostgreSQL...



Source: https://pub.intelligentx.net/book/export/html/554

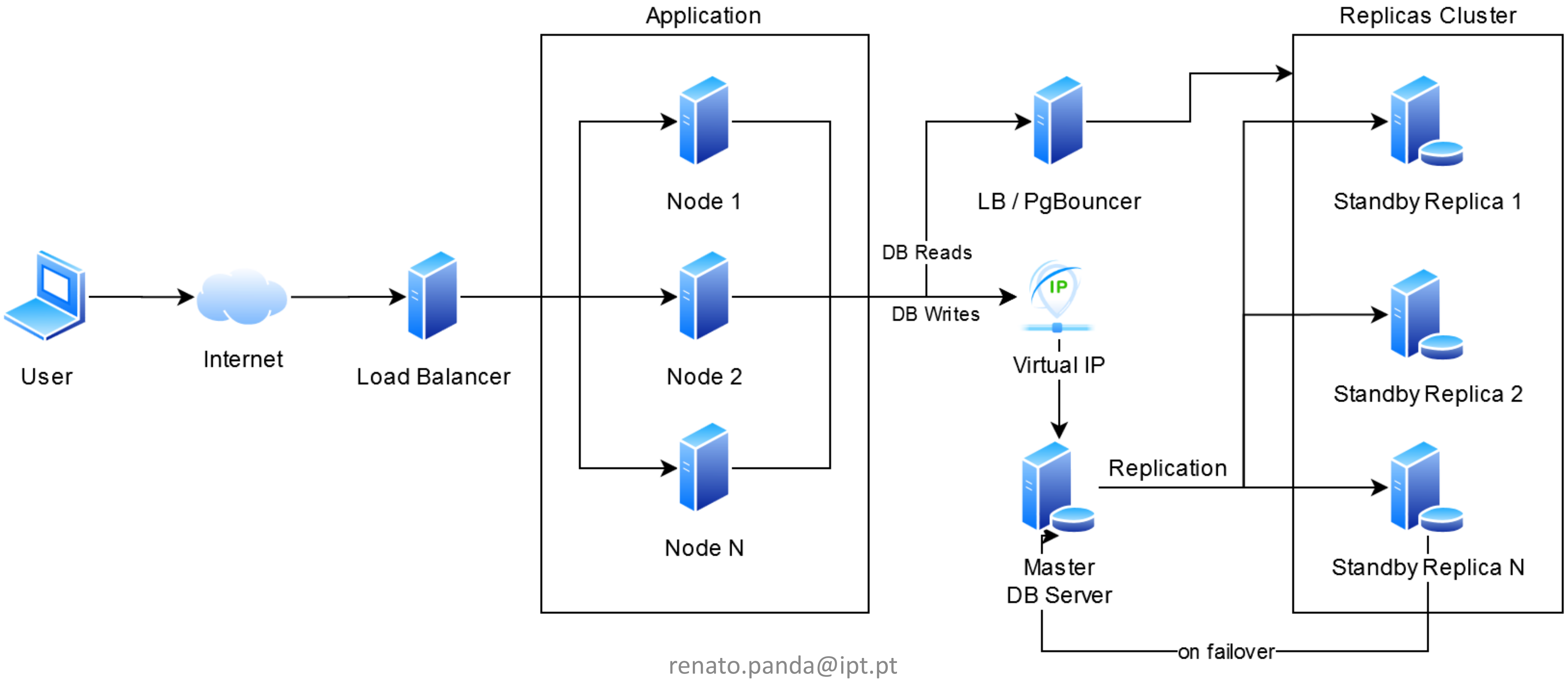
Problem #1: Database Layer

- What was achieved so far?
  - Setup of a master / replica (slave) PostgreSQL configuration



Problem #1: Database Layer

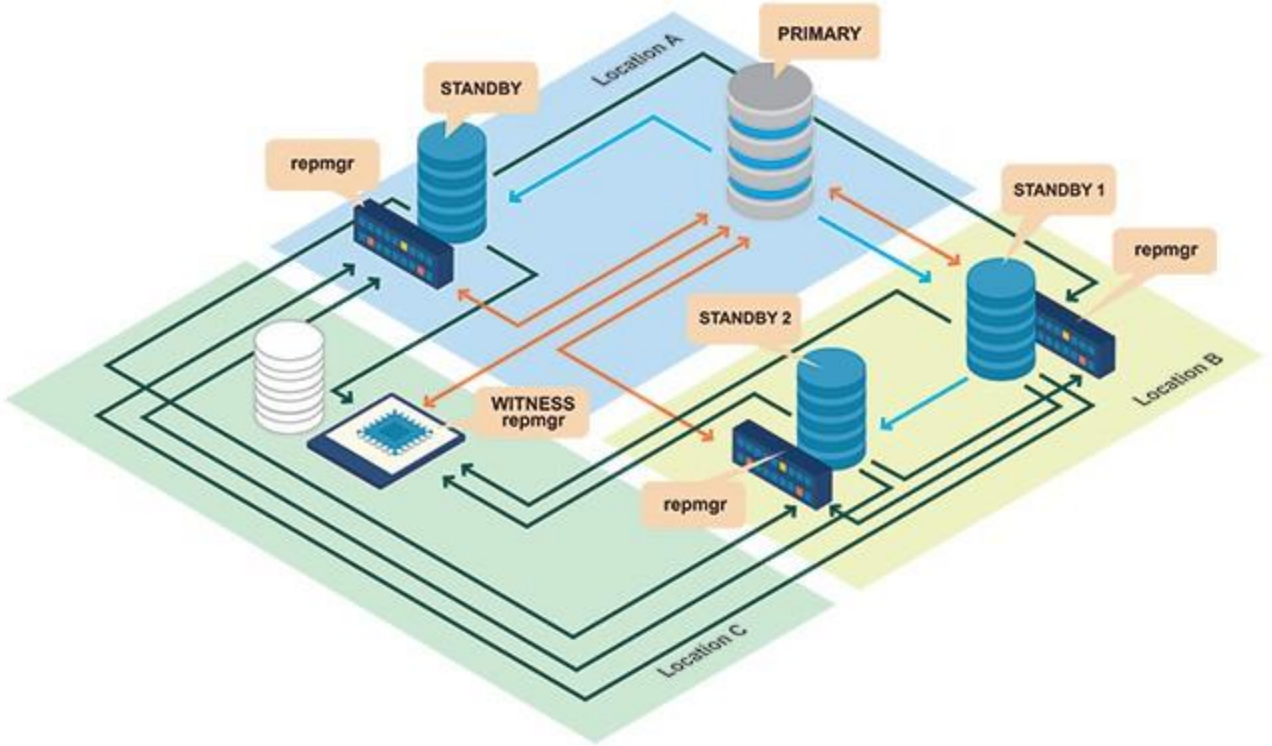
- What we really want in a real-life scenario?
  - Network-level (and so on) redundancy is obviously also needed



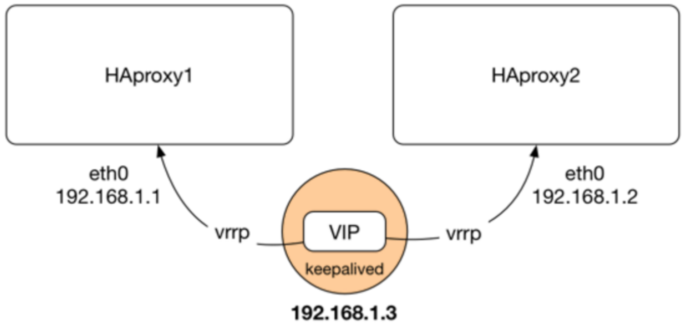
Problem #1: Database Layer

- What we really want in a real-life scenario?
  - Network-level (and so on) redundancy is obviously also needed

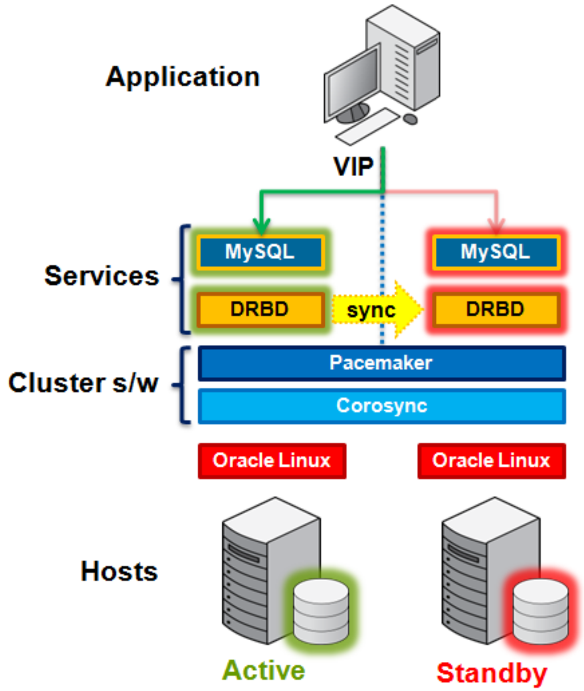
  - This is a simplified view
    - DB witness / observer
    - LB redundancy
    - Redundancy across sites
      - E.g., EU & US datacenters
    - Data storage?



Problem #1: Database Layer



- Other common HA & failover tools
  - Keepalived
    - Used to **monitor** services or systems and to **automatically failover** to a standby if problems occur
  - Corosync + Pacemaker
    - Corosync provides cluster membership and messaging capabilities to servers
    - Pacemaker is a cluster resource manager, provides the ability to control how the cluster behaves



Problem #1: Database Layer

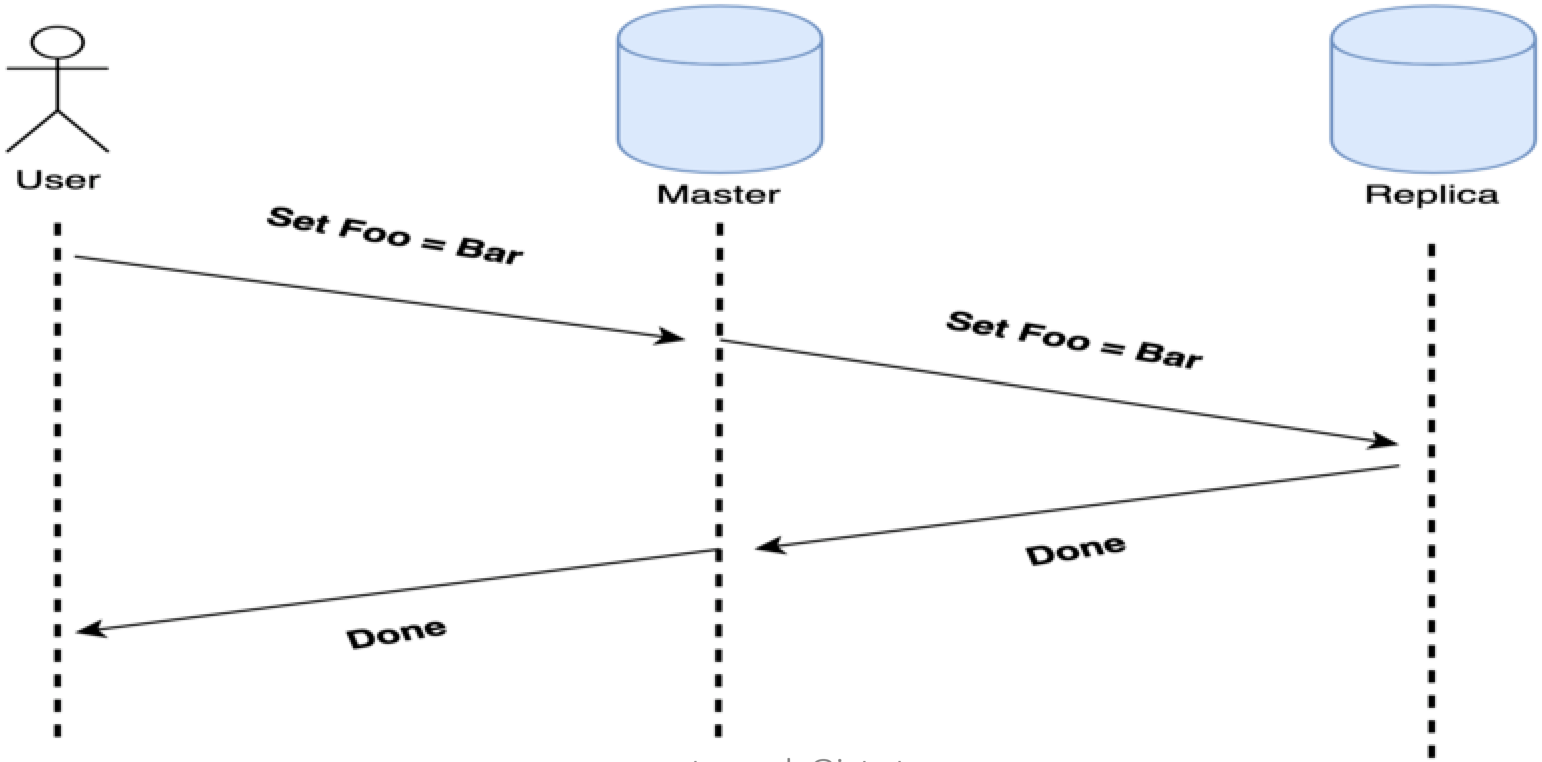
- There are different types of replication strategies:
  - Synchronous replication
  - Asynchronous replication
  - Semi-synchronous replication

Credits: https://thamaraiselvam.dev/synchronous-vs-asynchronous-database-replication-ck6i6zv3v00bld9s1c8e2vskc

Problem #1: Database Layer

- There are different types of replication strategies:
  - Synchronous replication

What can go wrong here?

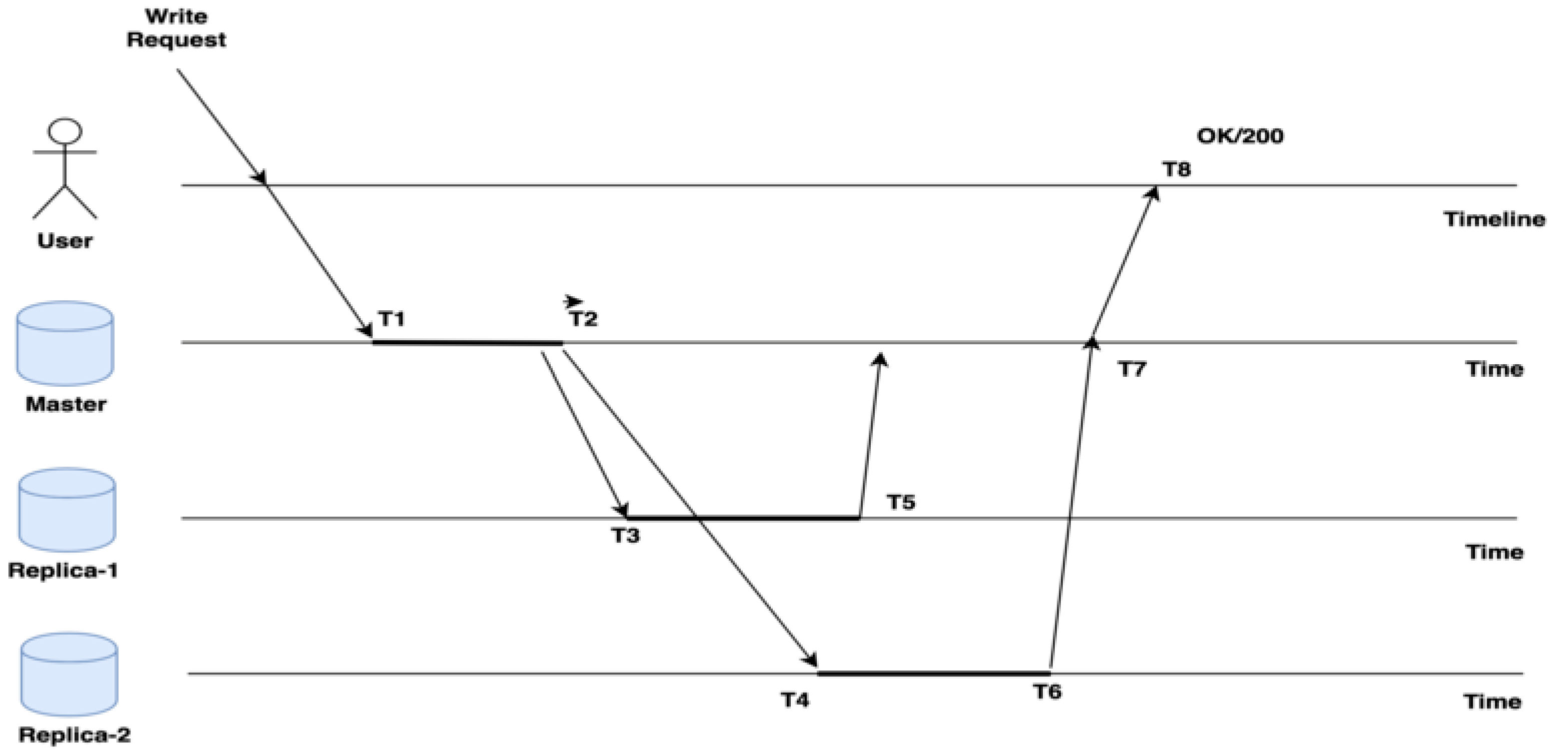


Credits: https://thamaraiselvam.dev/synchronous-vs-asynchronous-database-replication-ck6i6zv3v00bld9s1c8e2vskc

Problem #1: Database Layer

- There are different types of replication strategies:
  - Synchronous replication

What can go wrong here?

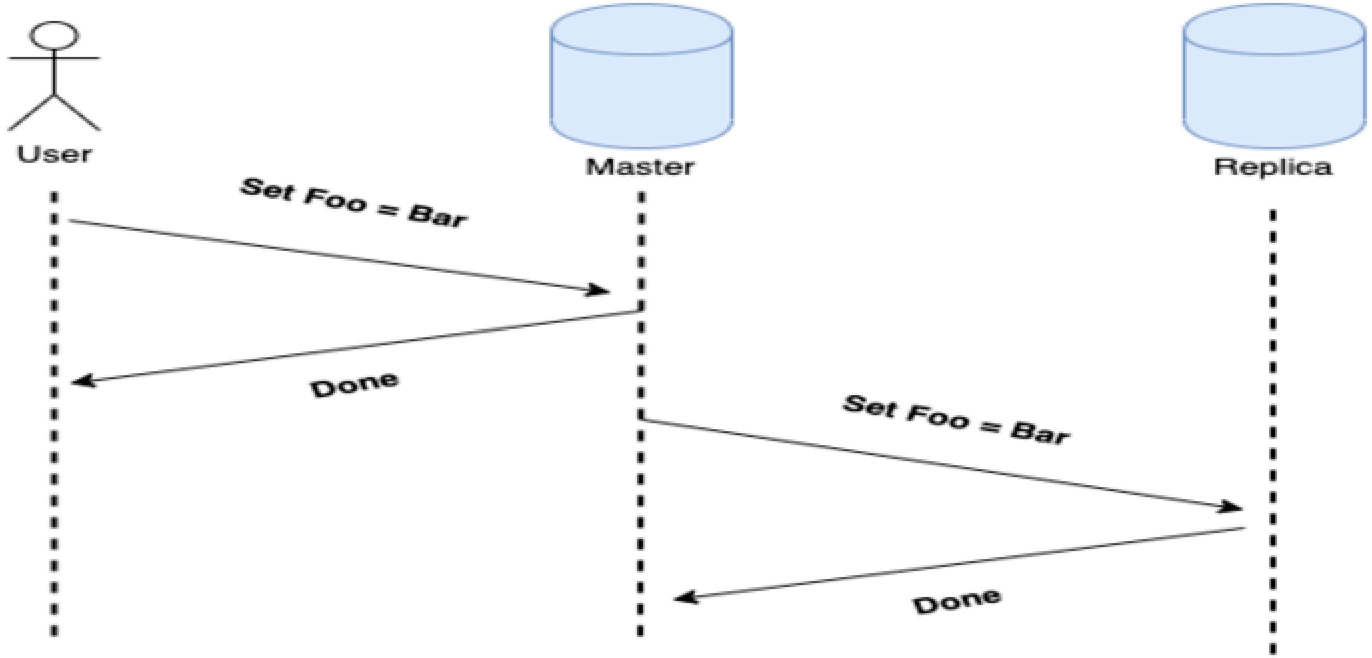


Credits: https://thamaraiselvam.dev/synchronous-vs-asynchronous-database-replication-ck6i6zv3v00bld9s1c8e2vskc

Problem #1: Database Layer

- There are different types of replication strategies:
  - Asynchronous replication

What can go wrong here?

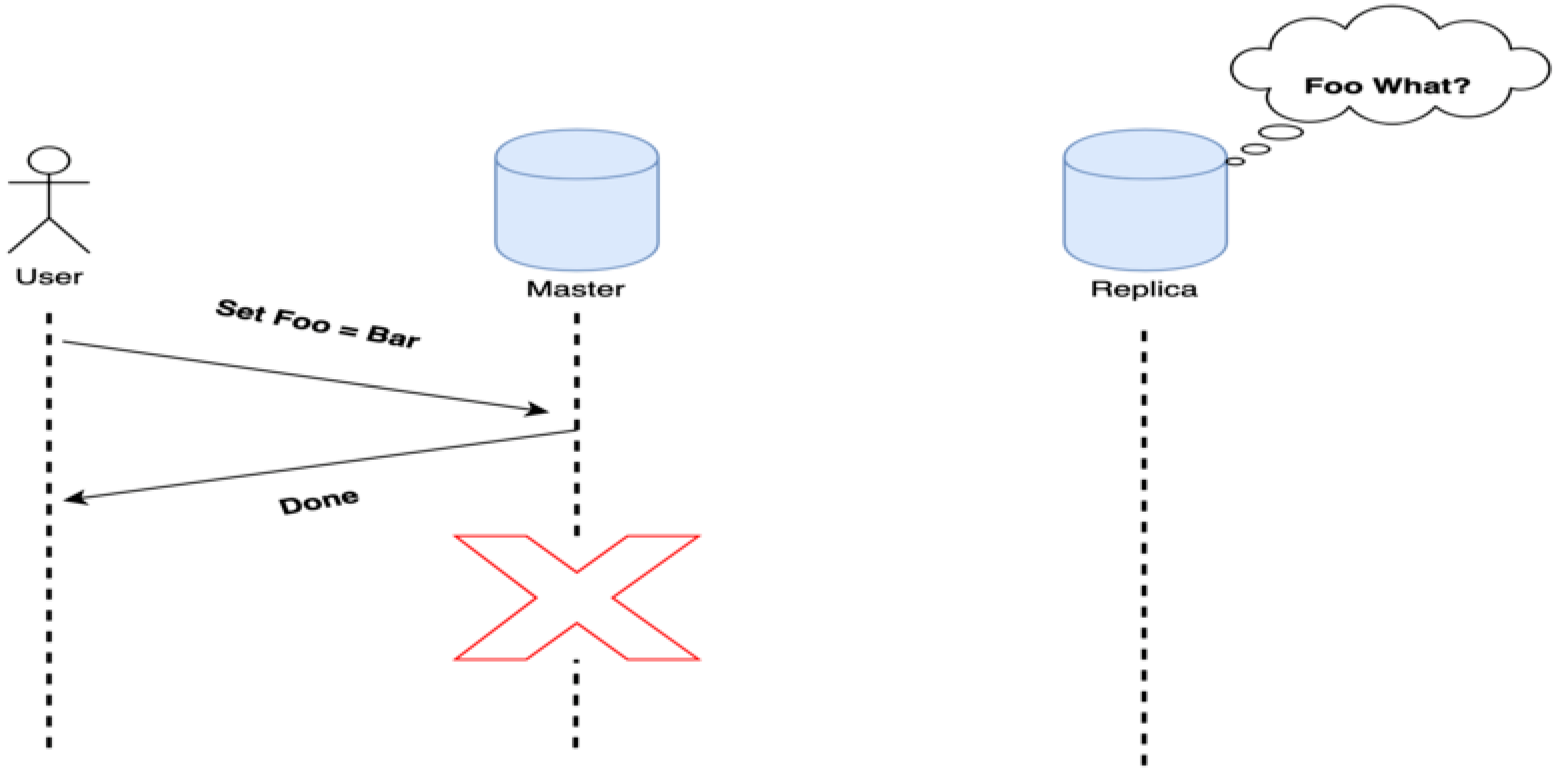


Credits: https://thamaraiselvam.dev/synchronous-vs-asynchronous-database-replication-ck6i6zv3v00bld9s1c8e2vskc

Problem #1: Database Layer

- There are different types of replication strategies:
  - Asynchronous replication

What can go wrong here?

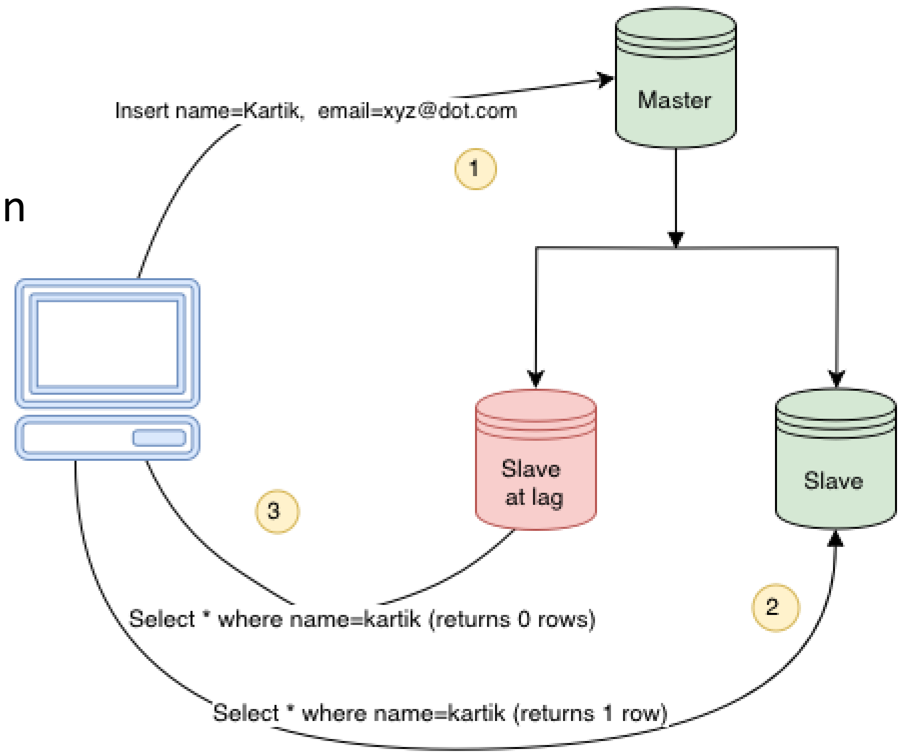


Credits: https://thamaraiselvam.dev/synchronous-vs-asynchronous-database-replication-ck6i6zv3v00bld9s1c8e2vskc

Problem #1: Database Layer

- There are different types of replication strategies:
  - Semi-synchronous replication

  - Middle ground approach
    - Some first level replicas have synchronous replication
    - The remaining ones use asynchronous replication



Other topics: multi-master replication, data partitioning, …

Credits: https://thamaraiselvam.dev/synchronous-vs-asynchronous-database-replication-ck6i6zv3v00bld9s1c8e2vskc

Problem #1: Database Layer

- 2nd attempt: use repmgr to manage DB clusters
  - Allows for master/replicas setup
  - Supports failover mechanisms (promote standby to primary)
    - Manual (switchover) or automatic (primary down)
  - Witness node can be used

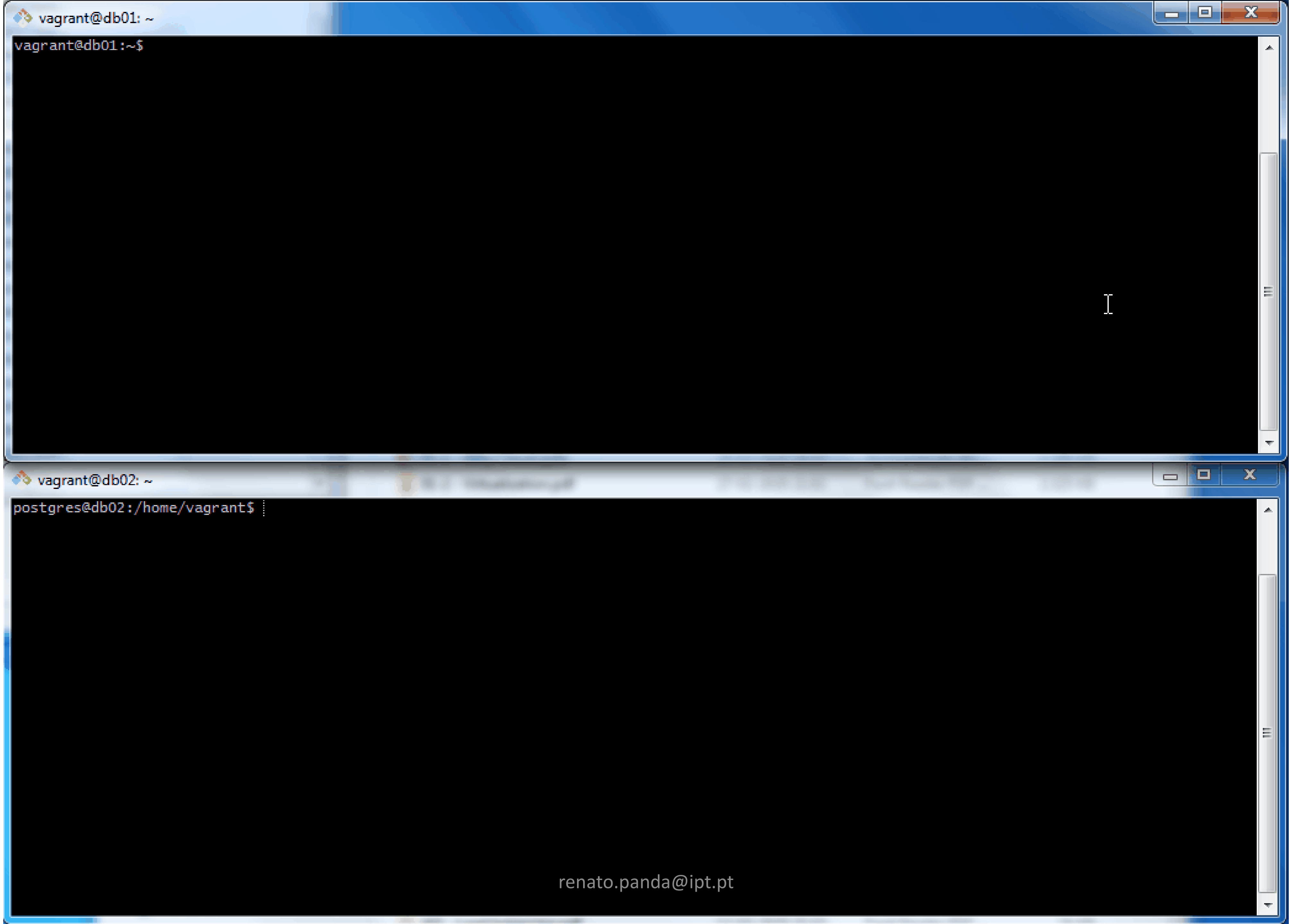
Problem #1: Database Layer

- 2nd attempt: simple repmgr primary/standby cluster example

```
Vagrant.configure("2") do |config|
  config.vm.define "db01" do |db01|
    db01.vm.box = "bento/ubuntu-16.04"
    db01.vm.hostname = "db01"
    db01.vm.network :private_network, ip: "192.168.33.150"
    db01.vm.provider "virtualbox" do |v|
      v.memory = 512
    end
    db01.vm.provision "shell", path: "install_postgres.sh"
    db01.vm.provision "shell", path: "install_repmgr.sh"
    db01.vm.provision "shell", path: "setup_primary.sh"
  end

  config.vm.define "db02" do |db02|
    db02.vm.box = "bento/ubuntu-16.04"
    db02.vm.hostname = "db02"
    db02.vm.network :private_network, ip: "192.168.33.151"
    db02.vm.provider "virtualbox" do |v|
      v.memory = 512
    end
    db02.vm.provision "shell", path: "install_postgres.sh"
    db02.vm.provision "shell", path: "install_repmgr.sh"
    db02.vm.provision "shell", path: "setup_standby.sh"
    # note use vagrant/shell args to pass IP as param and setup N standby servers
  end
```

Show scripts
Migrate to Ansible


vagrant@db01: ~
vagrant@db01:~$
vagrant@db02: ~
postgres@db02:/home/vagrant$

vagrant@db01: ~

```
postgres@db01:/home/vagrant$ repmgr cluster show
 ID | Name | Role    | Status    | Upstream | Location | Priority | Timeline | Connection string
----+------+---------+-----------+----------+----------+----------+----------+------------------------------------------------------------------
 1  | db01 | primary | * running |          | default  | 100      | 3        | host=192.168.33.150 user=repmgr dbname=repmgr connect_timeout=2
 2  | db02 | standby |   running | db01     | default  | 100      | 3        | host=192.168.33.151 user=repmgr dbname=repmgr connect_timeout=2
postgres@db01:/home/vagrant$
```

vagrant@db02: ~

```
postgres@db02:/home/vagrant$ repmgr cluster show
 ID | Name | Role    | Status    | Upstream | Location | Priority | Timeline | Connection string
----+------+---------+-----------+----------+----------+----------+----------+------------------------------------------------------------------
 1  | db01 | primary | * running |          | default  | 100      | 3        | host=192.168.33.150 user=repmgr dbname=repmgr connect_timeout=2
 2  | db02 | standby |   running | db01     | default  | 100      | 3        | host=192.168.33.151 user=repmgr dbname=repmgr connect_timeout=2
postgres@db02:/home/vagrant$ repmgr standby switchover --siblings-follow --dry-run
NOTICE: checking switchover on node "db02" (ID: 2) in --dry-run mode
INFO: SSH connection to host "192.168.33.150" succeeded
INFO: able to execute "repmgr" on remote host "localhost"
WARNING: option "--sibling-nodes" specified, but no sibling nodes exist
INFO: 1 walsenders required, 10 available
INFO: demotion candidate is able to make replication connection to promotion candidate
INFO: 0 pending archive files
INFO: replication lag on this standby is 0 seconds
NOTICE: local node "db02" (ID: 2) would be promoted to primary; current primary "db01" (ID: 1) would be demoted to standby
INFO: following shutdown command would be run on node "db01":
  "sudo pg_ctlcluster 12 main stop"
INFO: prerequisites for executing STANDBY SWITCHOVER are met
postgres@db02:/home/vagrant$
```

vagrant@db01: ~

```
postgres@db01:~$ repmgr cluster show
 ID | Name | Role    | Status    | Upstream | Location | Priority | Timeline | Connection string
----+------+---------+-----------+----------+----------+----------+----------+------------------------------------------------------------------------
 1  | db01 | primary | * running |          | default  | 100      | 3        | host=192.168.33.150 user=repmgr dbname=repmgr connect_timeout=2
 2  | db02 | standby |   running | db01     | default  | 100      | 3        | host=192.168.33.151 user=repmgr dbname=repmgr connect_timeout=2
postgres@db01:~$
```

vagrant@db02: ~

```
postgres@db02:~$
```

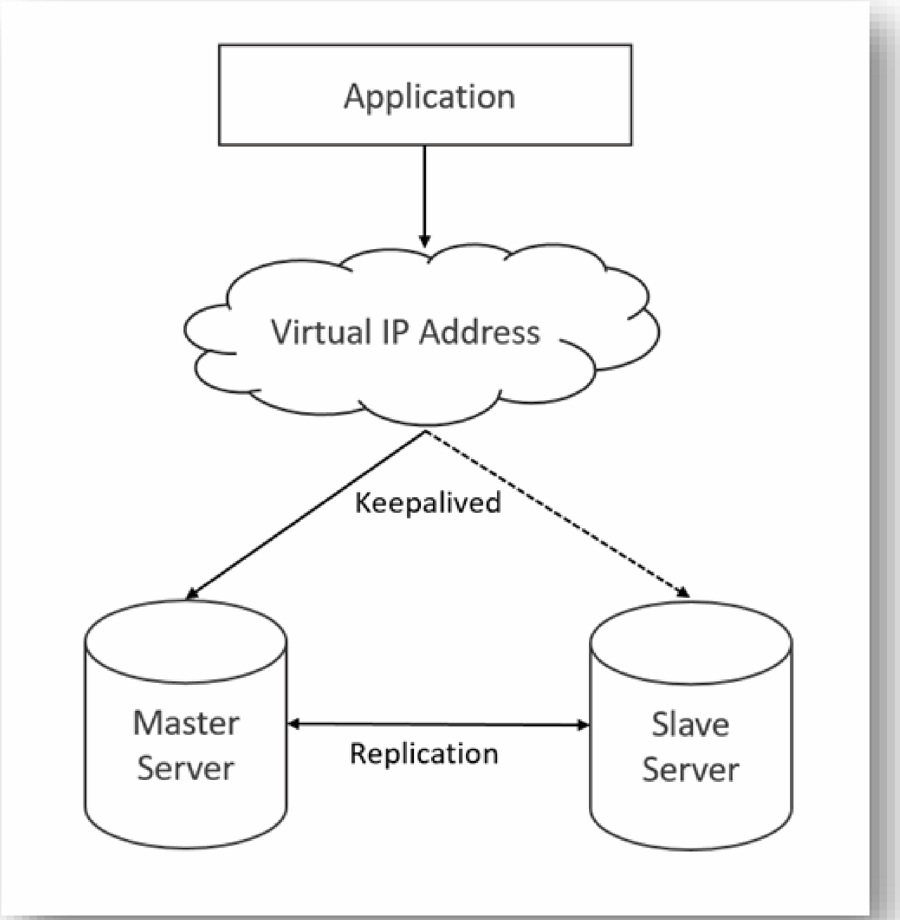
Problem #1: Database Layer

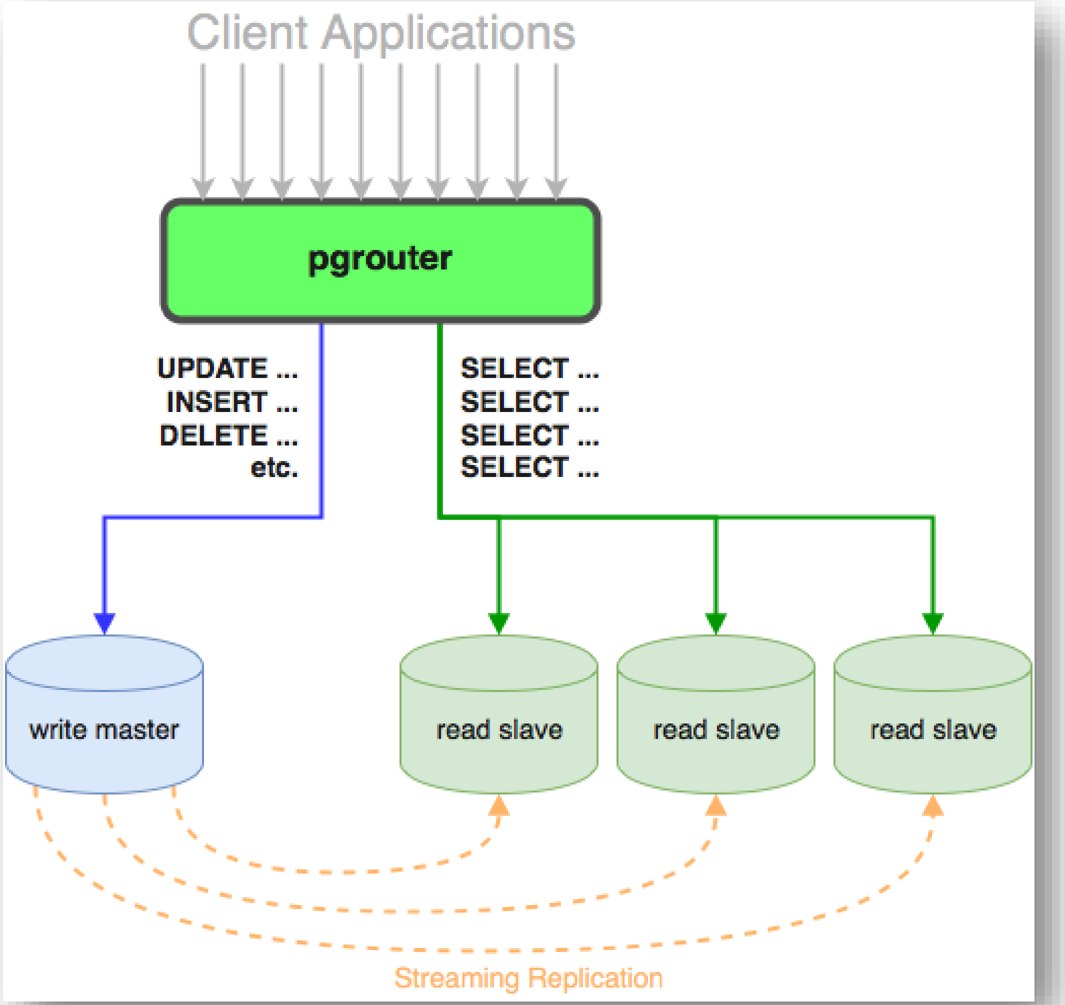
- What happened?
  - Two DB servers setup with repmgr
    - Primary (read/writes) and standby (read-only replica)
  - Switchover setup and test
    - Standby can be manually promoted to primary via switchover

- What is still missing (next year?)
  - Automatic promotion on failover
  - Witness setup to achieve quorum
  - Fencing or Virtual IP via PgBouncer

- If you like cool stories: https://about.gitlab.com/blog/2017/02/10/postmortem-of-database-outage-of-january-31/

Problem #2: Data Layer





Other approach: https://guides.rubyonrails.org/active_record_multiple_databases.html

Problem #1: Database Layer

This turned out to be faaaaar more complex than I thought!!

Do I really need all of these tools?

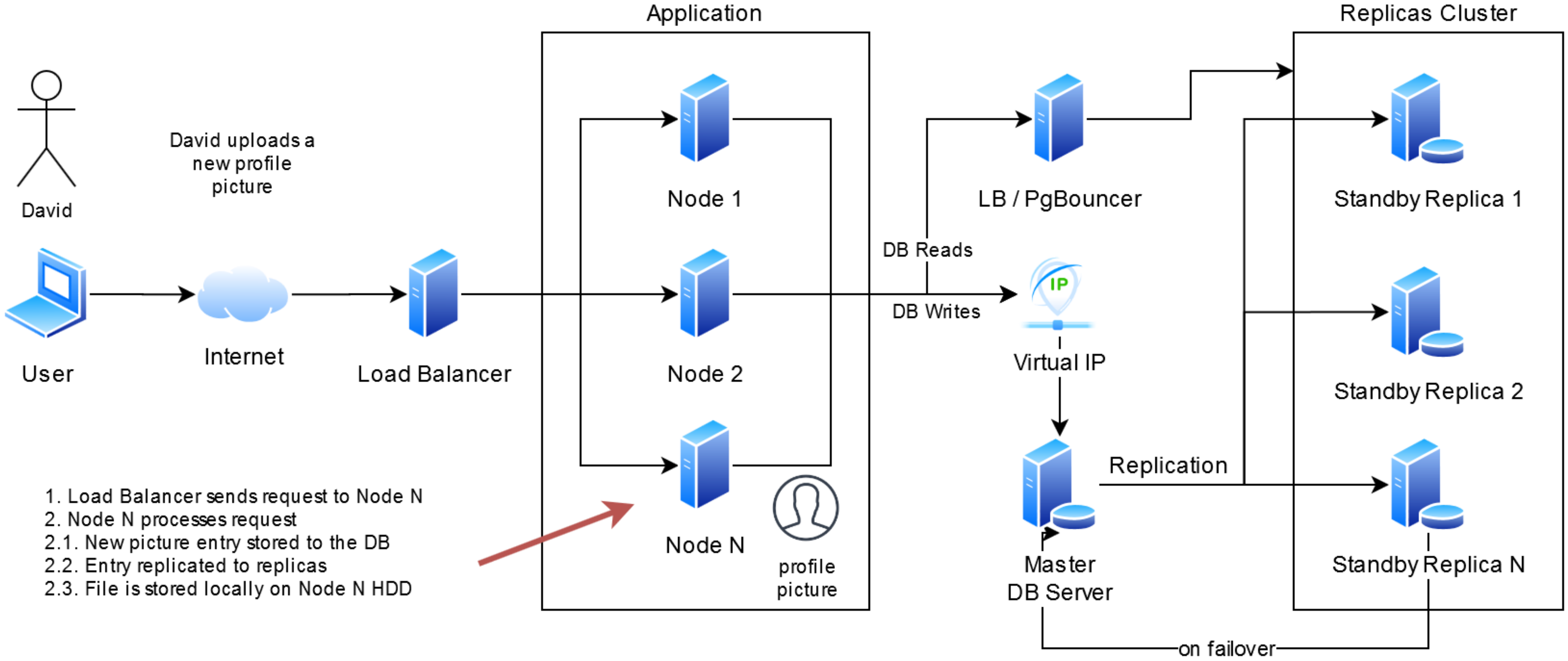
Can't I just use one single, simple, MySQL server just like in my <project/course>?

Problem #2: Data Layer

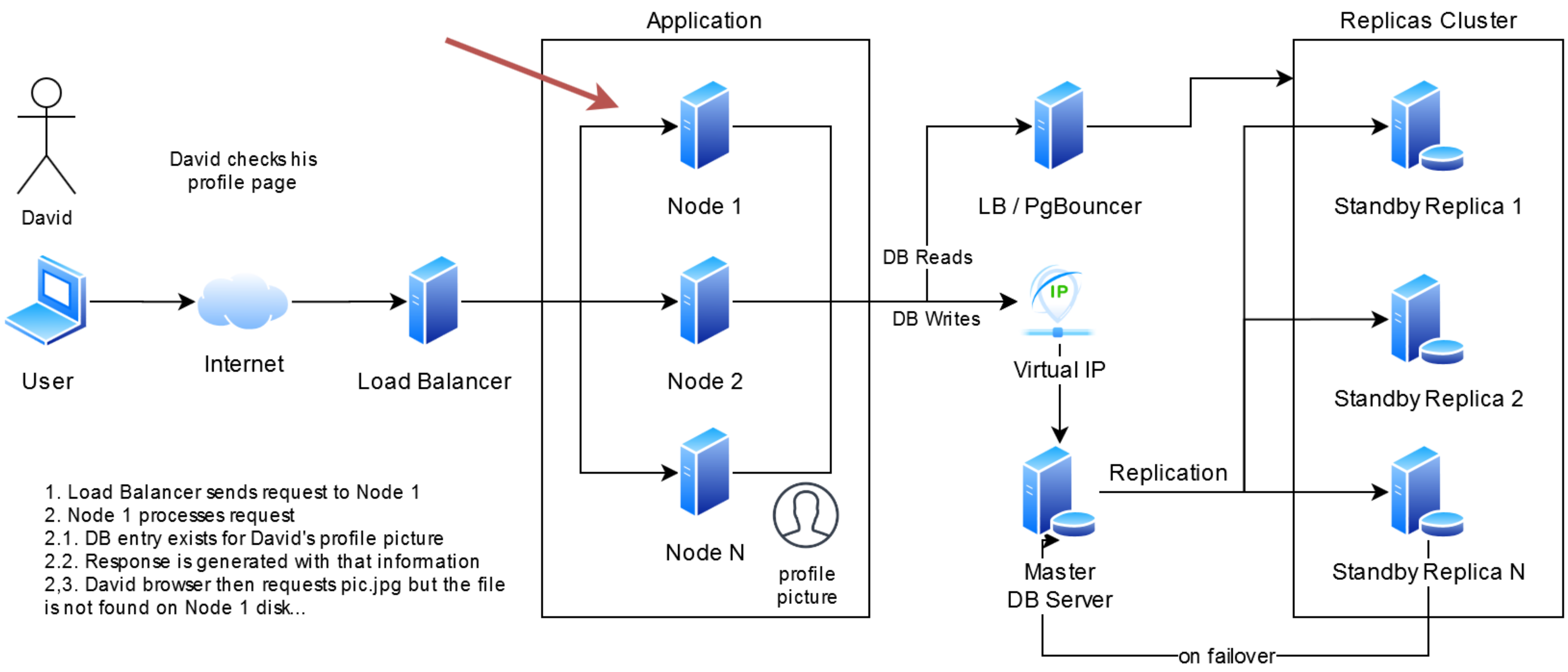
What about other types of data?

For instance, files uploaded by users are normally saved to the server HDD…

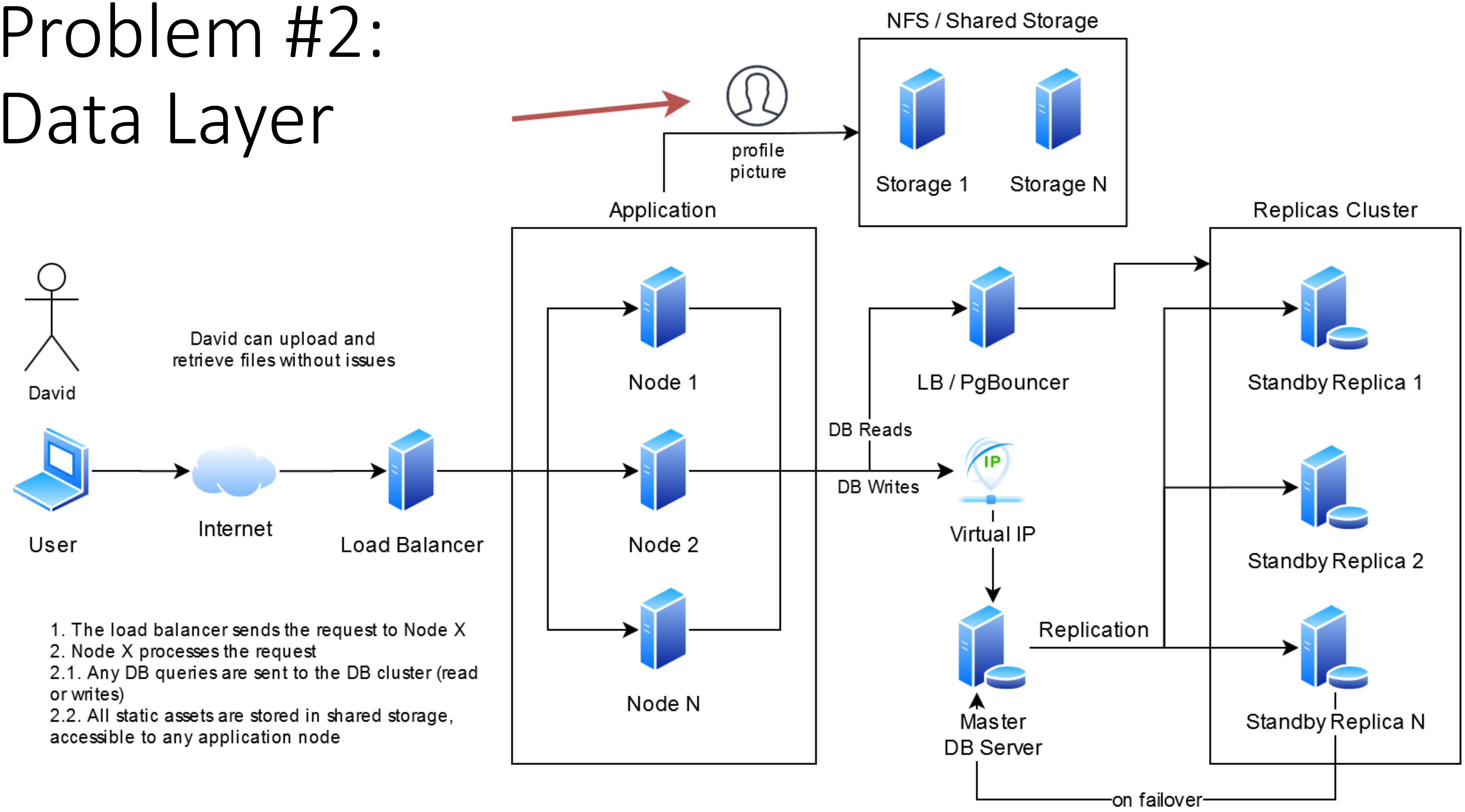
Problem #2: Data Layer



Problem #2: Data Layer



Problem #2: Data Layer



GitLab: Self-Managed Instance Scaling and High Availability

- GitLab - web-based DevOps lifecycle tool
  - Provides a Git-repository manager
    - Wiki
    - Issue-tracking
    - CI/CD pipeline features
  - Is open source
  - Initially in Ruby on Rails, current stack includes Go, Rails and Vue.js

  - Can be **hosted in-house**

GitLab: Self-Managed Instance Scaling and High Availability

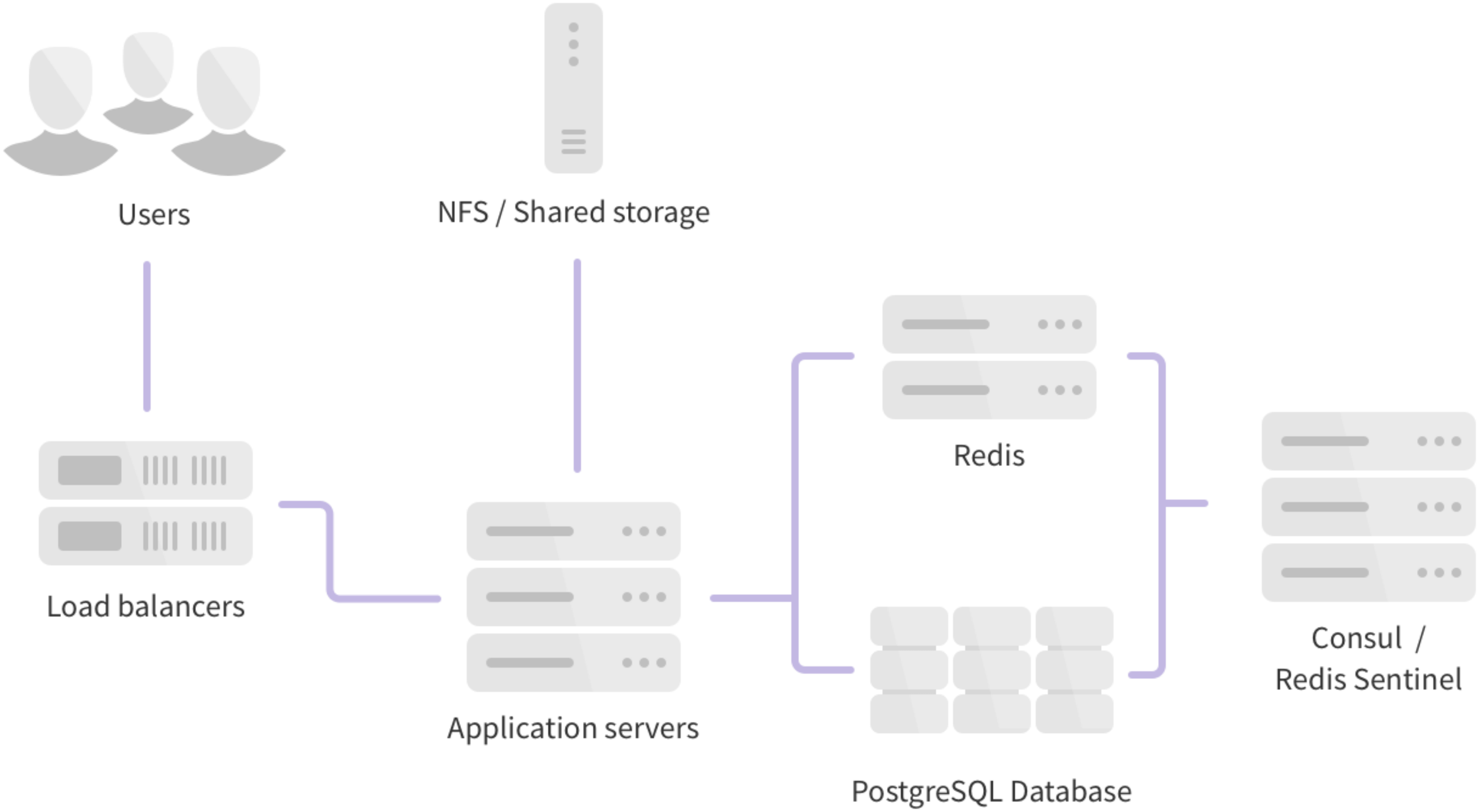
- GitLab [Administration Readme]
  - Contains guides for HA in self-hosted scenarios

“**Keep in mind that all highly-available solutions come with a trade-off between cost/complexity and uptime**. The more uptime you want, the more complex the solution. And the more complex the solution, the more work is involved in setting up and maintaining it. High availability is not free and every HA solution should balance the costs against the benefits.”

GitLab: Self-Managed Instance Scaling and High Availability

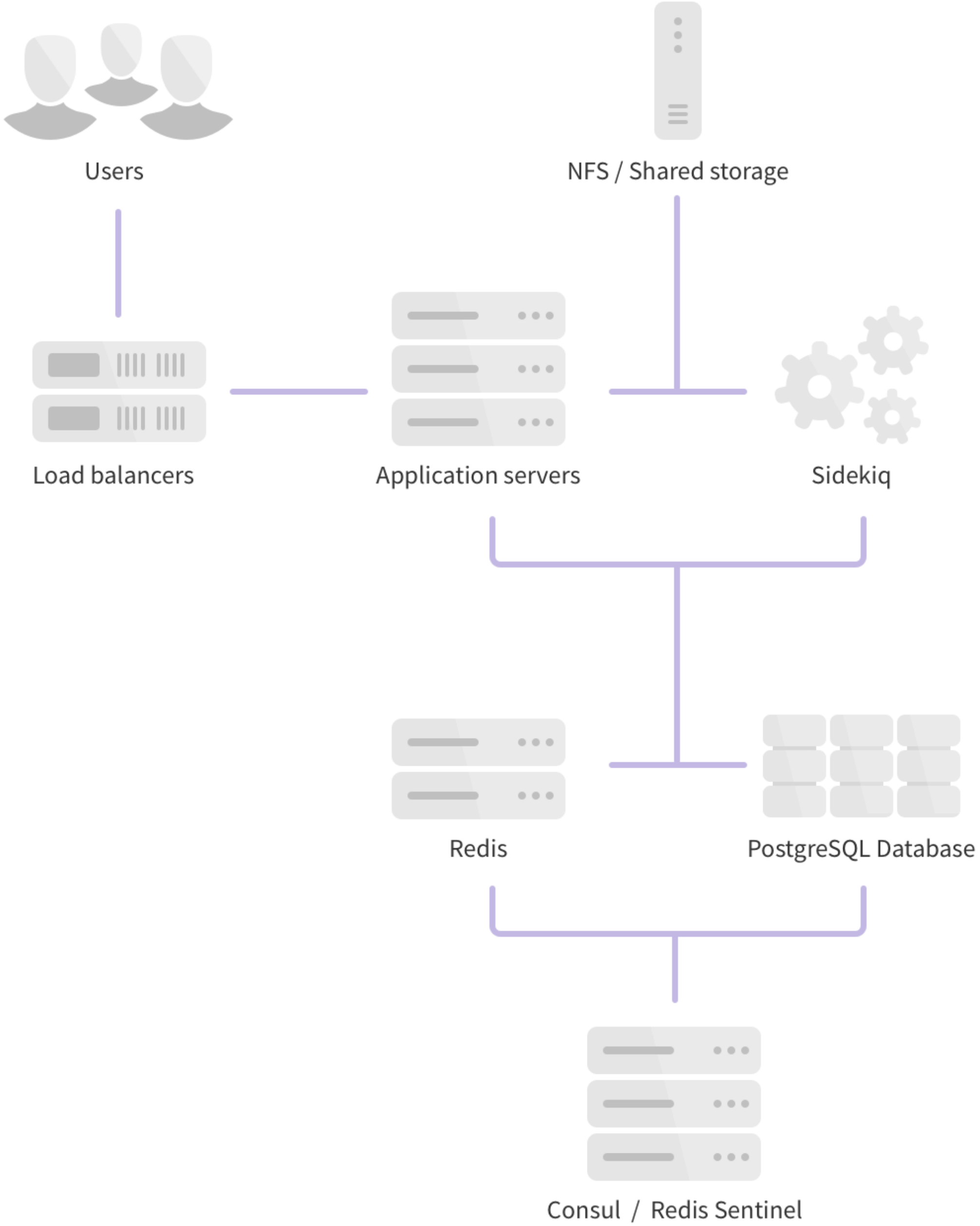
- GitLab Components (considered for a scaled or highly-available environment)
  - GitLab application nodes (Unicorn / Puma, Workhorse) - Web-requests (UI, API, Git over HTTP)
  - Sidekiq - Asynchronous/Background jobs
  - PostgreSQL - Database
    - Consul - Database service discovery and health checks/failover
    - PgBouncer - Database pool manager
  - Redis - Key/Value store (User sessions, cache, queue for Sidekiq)
    - Sentinel - Redis health check/failover manager
  - Gitaly - Provides high-level storage and RPC access to Git repositories
  - NFS storage servers (and / or S3 Object Storage service) for entities such as Uploads, Artifacts...
  - Load Balancer - Main entry point and handles load balancing for the GitLab application nodes
  - Monitor - Prometheus and Grafana monitoring with auto discovery.

GitLab: Scalable Architecture Examples – Horizontal

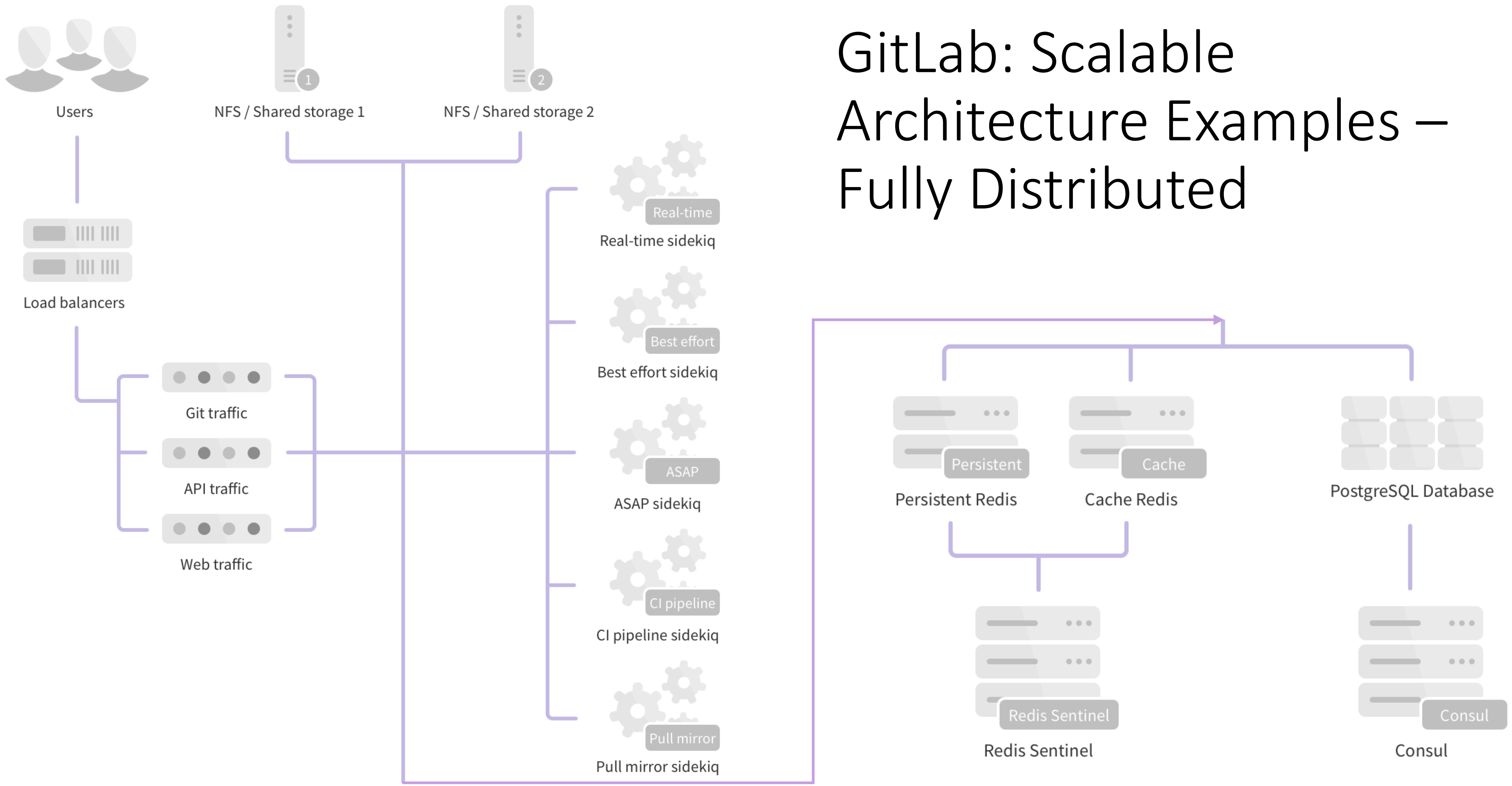


GitLab: Scalable Architecture Examples – Hybrid

In this architecture, certain components are split on dedicated nodes so high resource usage of one component does not interfere with others.



GitLab: Scalable Architecture Examples – Fully Distributed



GitLab Reference Architecture Recommendations

2,000 User Configuration

- **Supported Users (approximate):** 2,000
- **Test RPS Rates:** API: 40 RPS, Web: 4 RPS, Git: 4 RPS
- **Known Issues:** For the latest list of known performance issues head here.

Service	Nodes	Configuration	GCP type
GitLab Rails[4]	3	8 vCPU, 7.2GB Memory	n1-highcpu-8
PostgreSQL	3	2 vCPU, 7.5GB Memory	n1-standard-2
PgBouncer	3	2 vCPU, 1.8GB Memory	n1-highcpu-2
Gitaly[5 6]	X	4 vCPU, 15GB Memory	n1-standard-4
Redis[7]	3	2 vCPU, 7.5GB Memory	n1-standard-2
Consul + Sentinel[7]	3	2 vCPU, 1.8GB Memory	n1-highcpu-2
Sidekiq	4	2 vCPU, 7.5GB Memory	n1-standard-2
S3 Object Storage[1]	-	-	-
NFS Server[2 6]	1	4 vCPU, 3.6GB Memory	n1-highcpu-4
Monitoring node	1	2 vCPU, 1.8GB Memory	n1-highcpu-2
External load balancing node[3]	1	2 vCPU, 1.8GB Memory	n1-highcpu-2
Internal load balancing node[3]	1	2 vCPU, 1.8GB Memory	n1-highcpu-2

50,000 User Configuration

- **Supported Users (approximate):** 50,000
- **Test RPS Rates:** API: 1000 RPS, Web: 100 RPS, Git: 100 RPS
- **Known Issues:** For the latest list of known performance issues head here.

Service	Nodes	Configuration	GCP type
GitLab Rails[4]	15	32 vCPU, 28.8GB Memory	n1-highcpu-32
PostgreSQL	3	16 vCPU, 60GB Memory	n1-standard-16
PgBouncer	3	2 vCPU, 1.8GB Memory	n1-highcpu-2
Gitaly[5 6]	X	64 vCPU, 240GB Memory	n1-standard-64
Redis[7] - Cache	3	4 vCPU, 15GB Memory	n1-standard-4
Redis[7] - Queues / Shared State	3	4 vCPU, 15GB Memory	n1-standard-4
Redis Sentinel[7] - Cache	3	1 vCPU, 1.7GB Memory	g1-small
Redis Sentinel[7] - Queues / Shared State	3	1 vCPU, 1.7GB Memory	g1-small
Consul	3	2 vCPU, 1.8GB Memory	n1-highcpu-2
Sidekiq	4	4 vCPU, 15GB Memory	n1-standard-4
NFS Server[2 6]	1	4 vCPU, 3.6GB Memory	n1-highcpu-4
S3 Object Storage[1]	-	-	-
Monitoring node	1	4 vCPU, 3.6GB Memory	n1-highcpu-4
External load balancing node[3]	1	2 vCPU, 1.8GB Memory	n1-highcpu-2
Internal load balancing node[3]	1	8 vCPU, 7.2GB Memory	n1-highcpu-8

Problem #3: Balancing the Load Balancer?

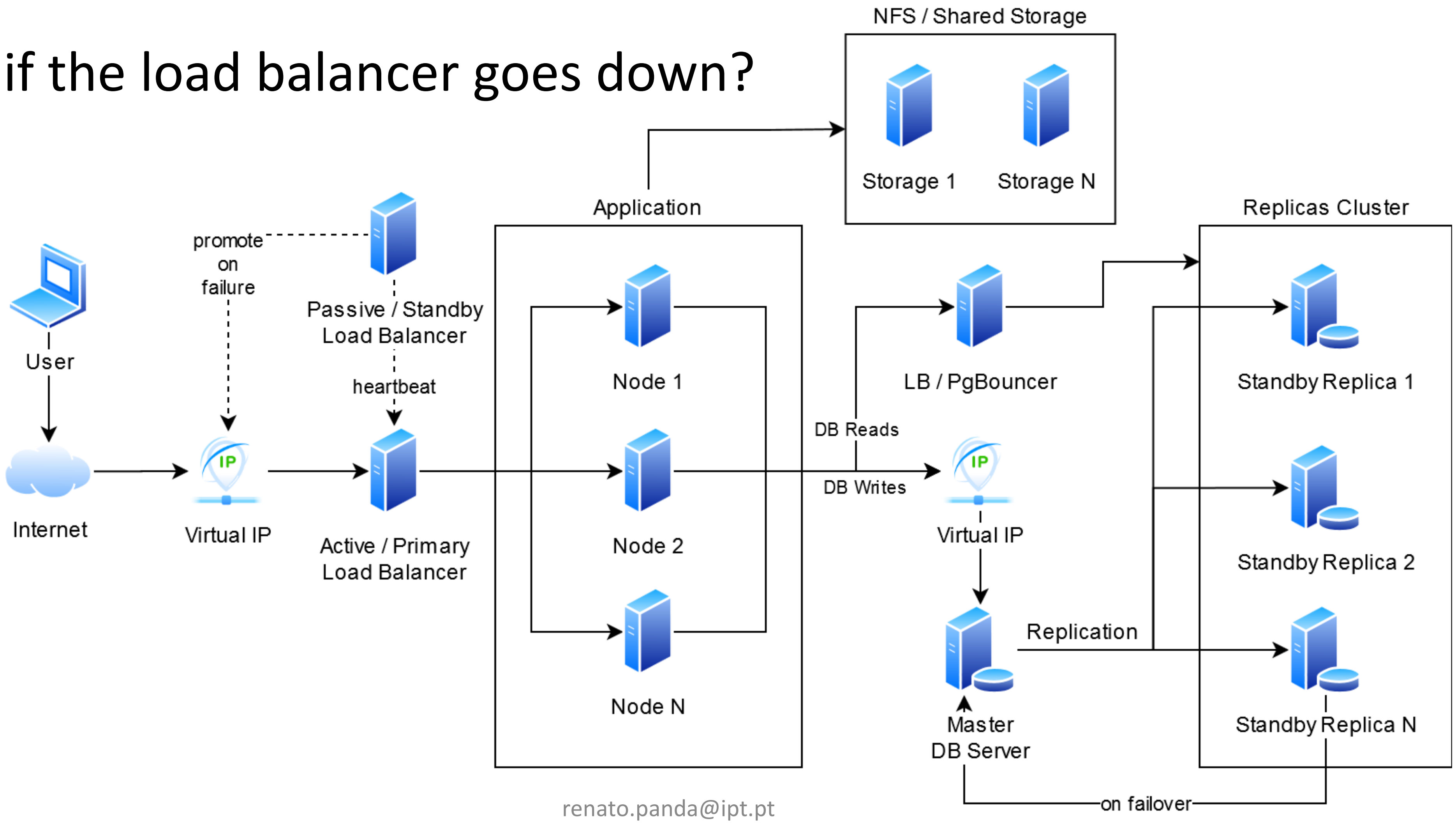
Okay! The database and storage parts make sense now…

… but what about the load balancer?

What if the LB goes down? What if it cannot handle all the requests?

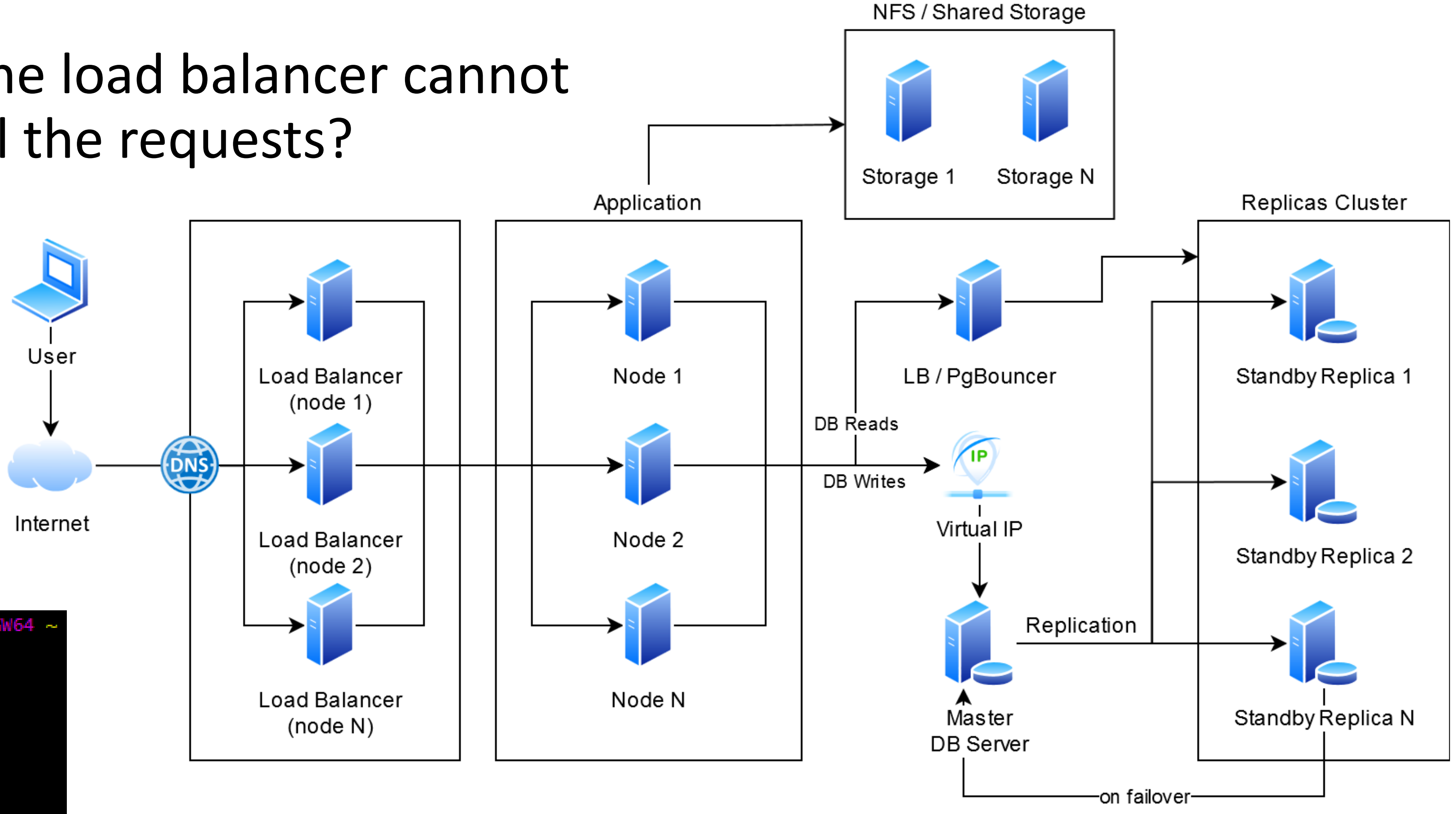
Problem #3: Balancing the Load Balancer?

- What if the load balancer goes down?



Problem #3: Balancing the Load Balancer?

- What if the load balancer cannot handle all the requests?



```
Renato Panda@panda-laptop MINGW64 ~
$ nslookup www.twitch.com
Resposta n▒o autoritativa:
Servidor:  vodafonegw
Address:  fe80::1

Nome:    www.twitch.com
Addresses:  52.38.159.119
          52.88.15.144
          52.24.254.52
```