LINUXPITER

11-12.11/2016

INTERNATIONAL LINUX CONFERENCE

ST.PETERSBURG

How devops exhausts itself, and what will happen next

Kirill Vechera
jetware.org

The next generation of models of software systems management

Kirill Vechera

CTO **jetware.org**

- Software system management automation
  - Building, configuring, testing, deployment, updating

Bio

- System programming
  - Solaris, FreeBSD, Linux
- System and network administration
  - Unix, OSPF, BGP
- Internet services development
  - C, Perl, Java, Python, Ruby

LINUX PITER

Annotation

The evolution of information systems management

Present instruments and emerging

How do we contribute to it

Why devops becomes unnecessary

LINUX PITER

Modern information system

Big

100 — 100 000 servers

Complex

10 — 100 apps

10 — 1000 connections

100 — 10 000 libraries and programs

Growing

0.1 — 100 updates/day

Problems

Control

Update

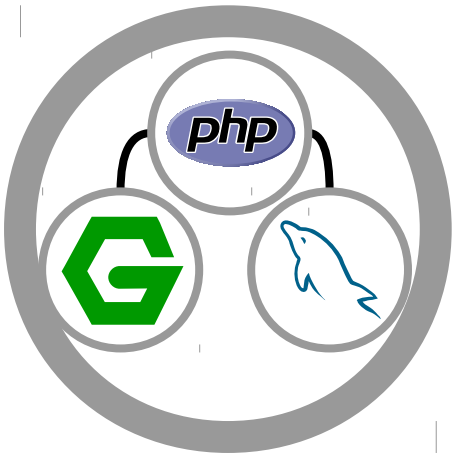
Diagnose

Optimize

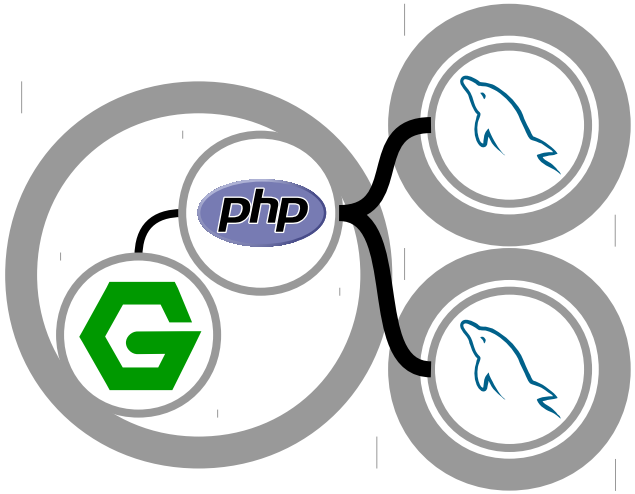
Repair

LINUX PITER

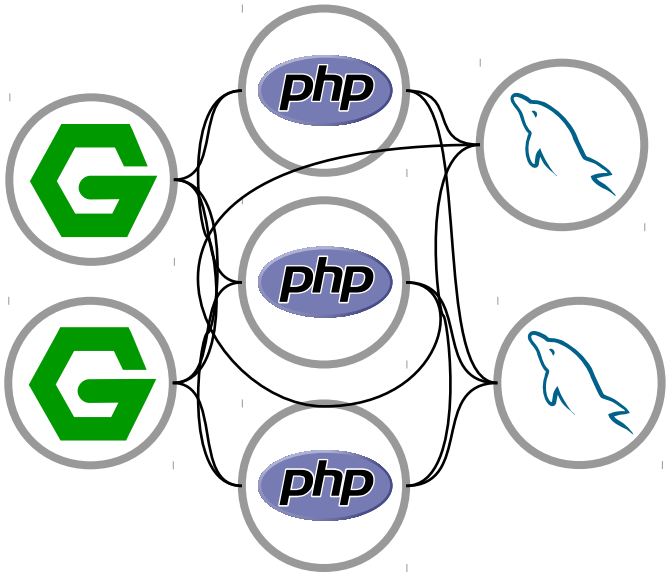
Structure of systems



Monolithic



Distributed,
tight coupled

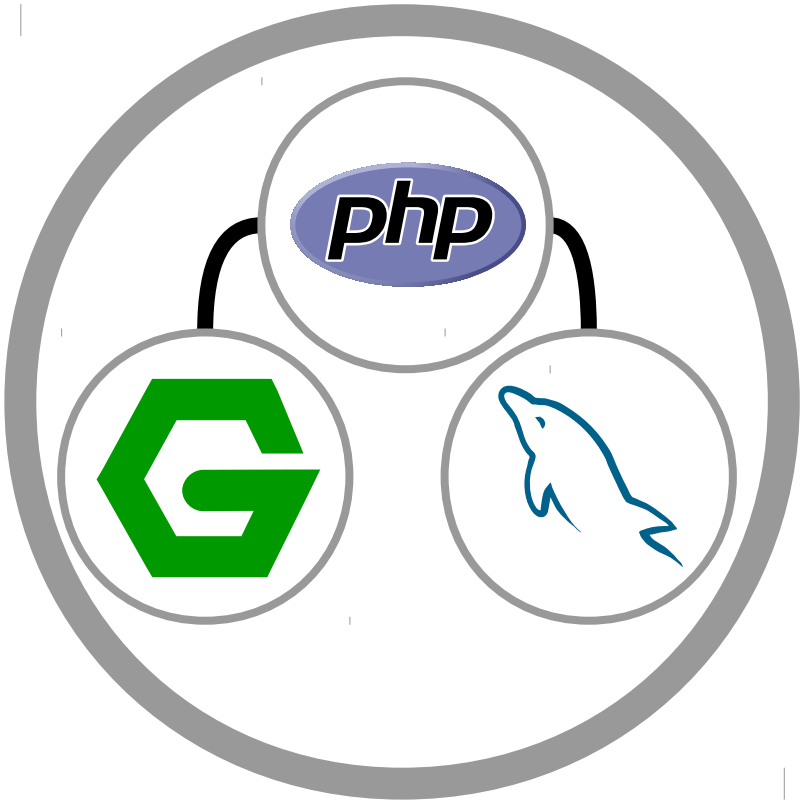


Distributed,
loosely coupled

LINUX PITER

Monolithic system

- Fixed configuration
- Everything as a single copy
  - Services
  - Connections
- Loss of service and communication disrupts the whole system
- Capacity is limited



LINUX PITER

Distributed system

Connected components

Component

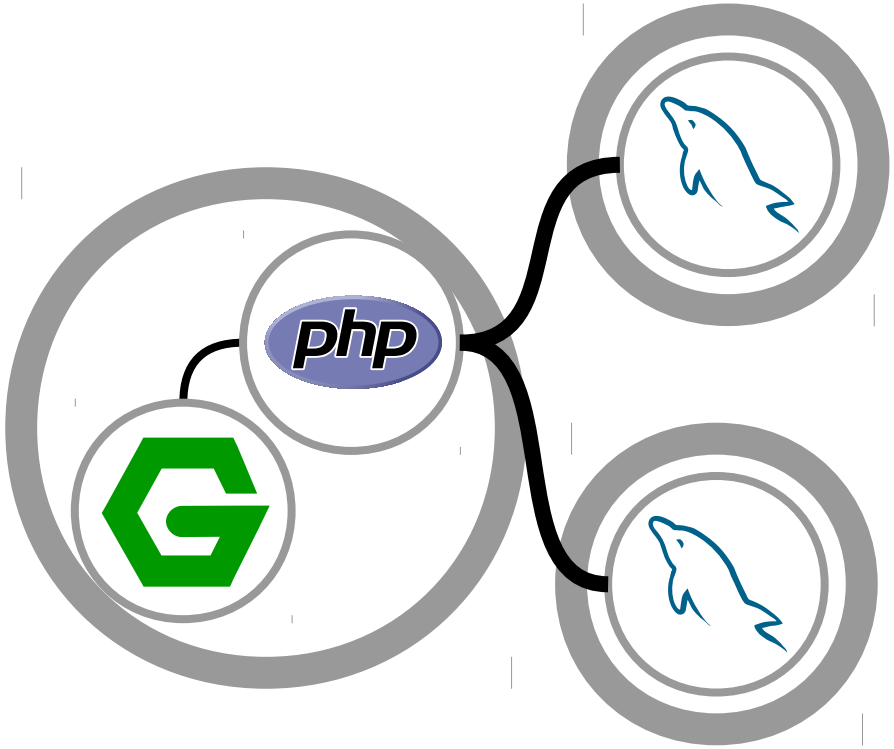
Single service or multiple different services together

Services

Single service or multiple instances of a service

Macrolevel

Microlevel



LINUX PITER

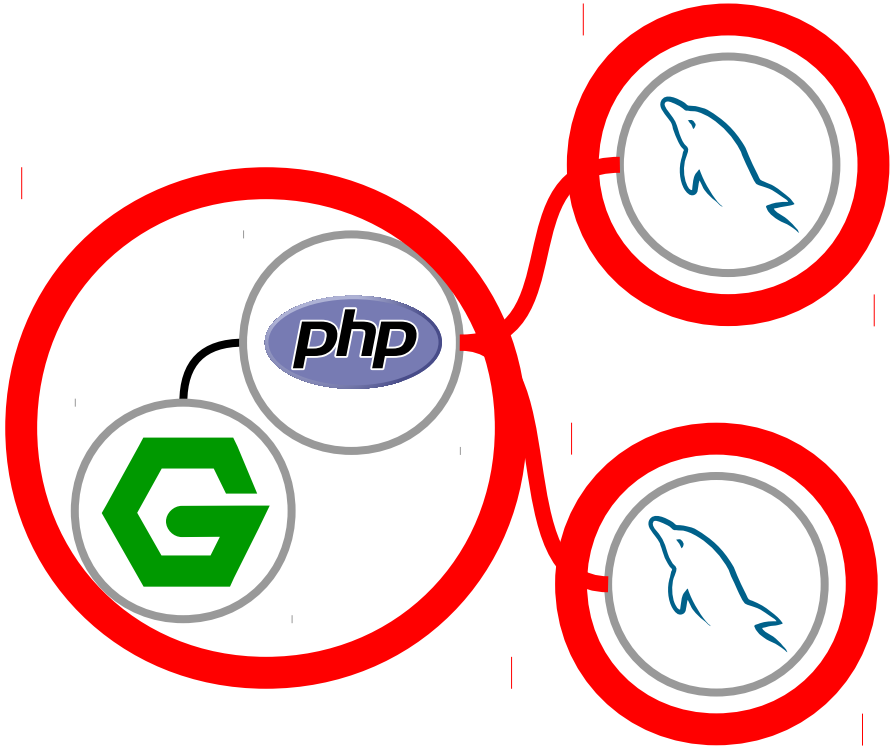
Macrolevel

Components

services

Topology

connections between the components

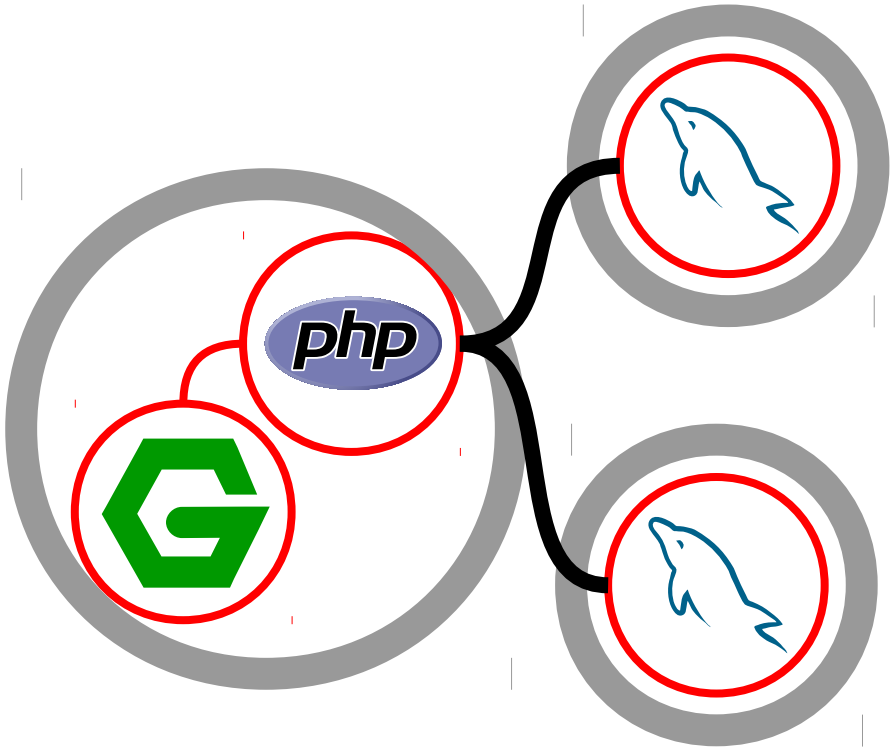


LINUX PITER

Microlevel

Internal arrangement of the component

- Programs, libraries, settings, data
- Connections between internal services



Inside the component — monolithic subsystem

LINUX PITER

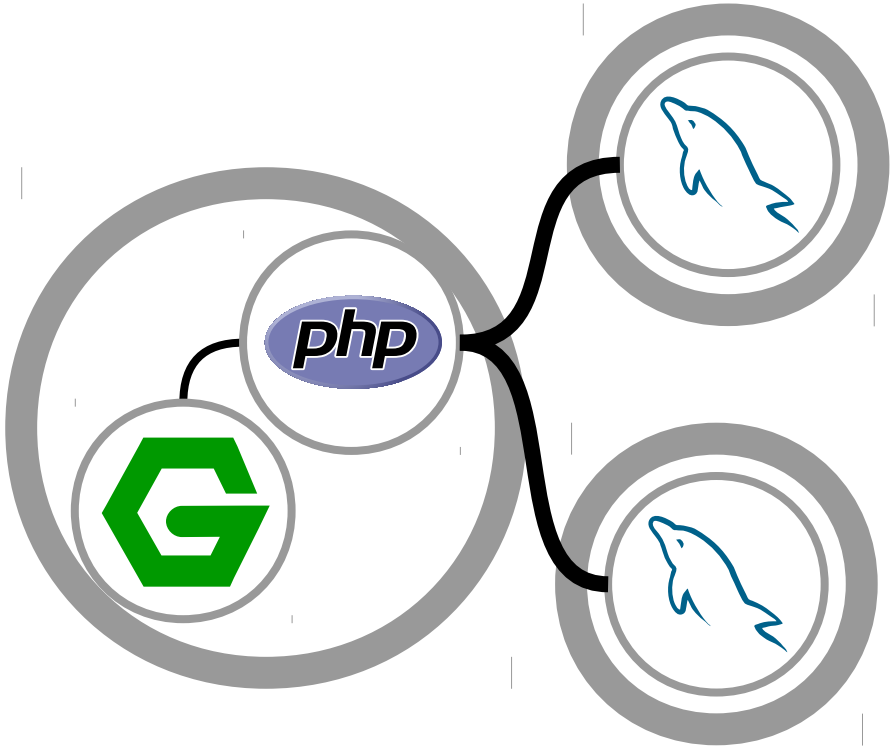
Distributed tightly coupled system

Everything is predetermined

- Connections to the services
- Binding to the equipment
- Adding/removing the components or connections
- The reaction to the failures

Management

- Centralized
- Connections and bindings are organized by the administrator



LINUX PITER

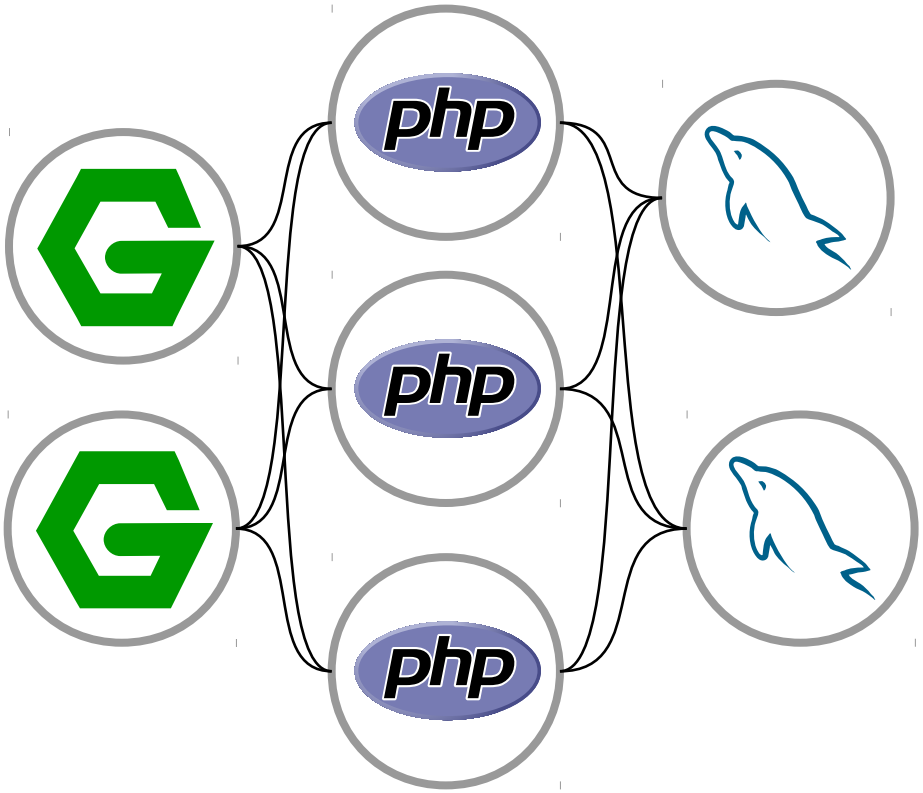
Distributed loosely coupled system

Everything is changeable

- Connections to the services
- Binding to the equipment
- Adding/removing the components or connections

Management

- Continuous
- Automated
- Centralized



LINUX PITER

Systems management evolution

Equally — both monolithic system, and tightly coupled distributed systems

3. Tools

2. Scripts

1. Hands

LINUX PITER

What follows IaC tools?

Switching to loosely coupled systems

Systems management is stratified

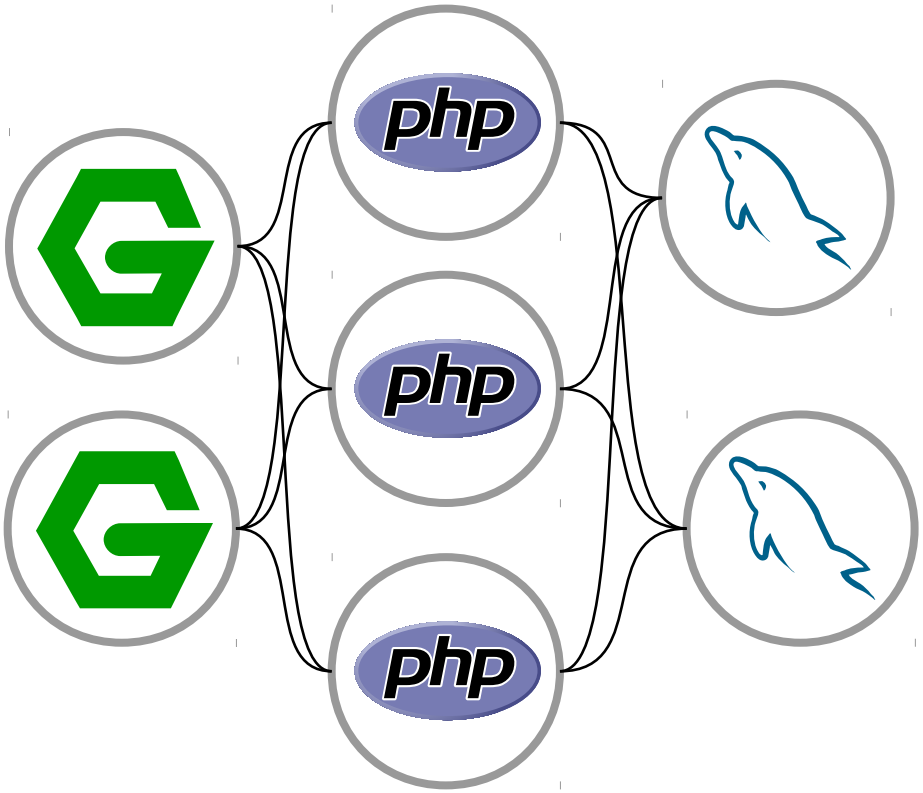
- We need macro-level menegement
- We need micro-level menegement

LINUX PITER

Loosely coupled system

Minimal requirements

- Automated management
  - Connections to the services
  - Binding to the equipment
  - Adding/removing the components or connections



LINUX PITER

Minimal requirements to the component

Portability

Replicability

Interface

Isolation from neighbours

Equipment independent

Separate date storage

Connections for services

LINUX PITER

AWS helped to get accustomed

Services oriented architechture

Immutable images of virtual machines

Separate data storage

Multiple service instances

Docker

PaaS basement

Development tool

Service execution engine

Docker-container as a component of a system

Lightweight replacement of the virtual machine

- Application isolation and dependency
- Interface - the network and data
  - Stimulates the use of immutable image

Dockerfile

- Fast building and replicability

All requirements for the component of loosely coupled system are complete

LINUX PITER

The evolution of distributed systems management – macrolevel

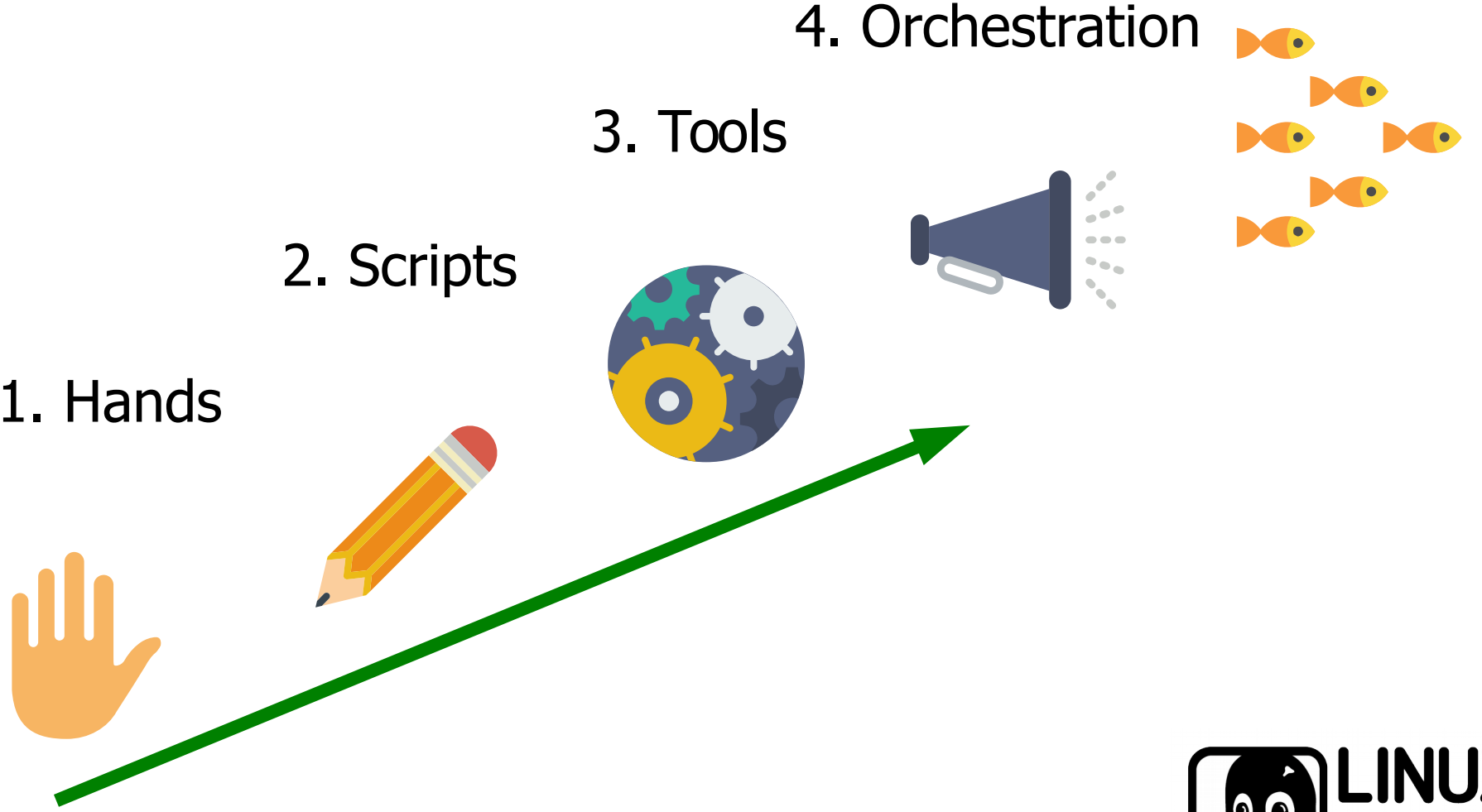
5. Self-organization

4. Orchestration

3. Tools

2. Scripts

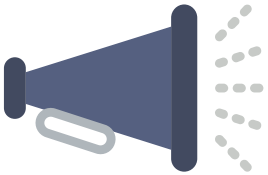
1. Hands



LINUX PITER

Distributed systems management

4. Orchestration



Alternatives

- Kubernetes
- Mesos Marathon
- Docker Swarm mode

LINUX PITER

The evolution – microlevel, monolithic system

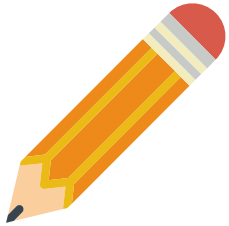
5. Self-organization

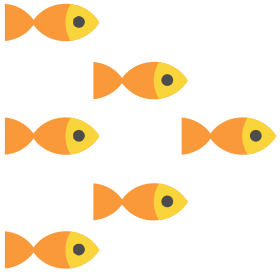
4. ???

3. Tools

2. Scripts

1. Hands





LINUX PITER

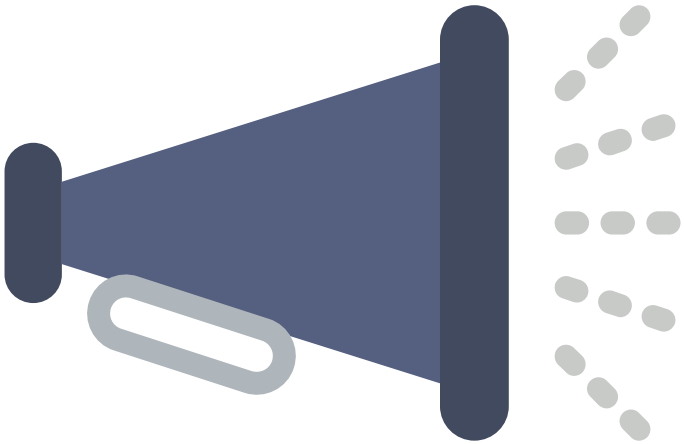
The evolution of distributed systems management

Macro-level

Orchestration

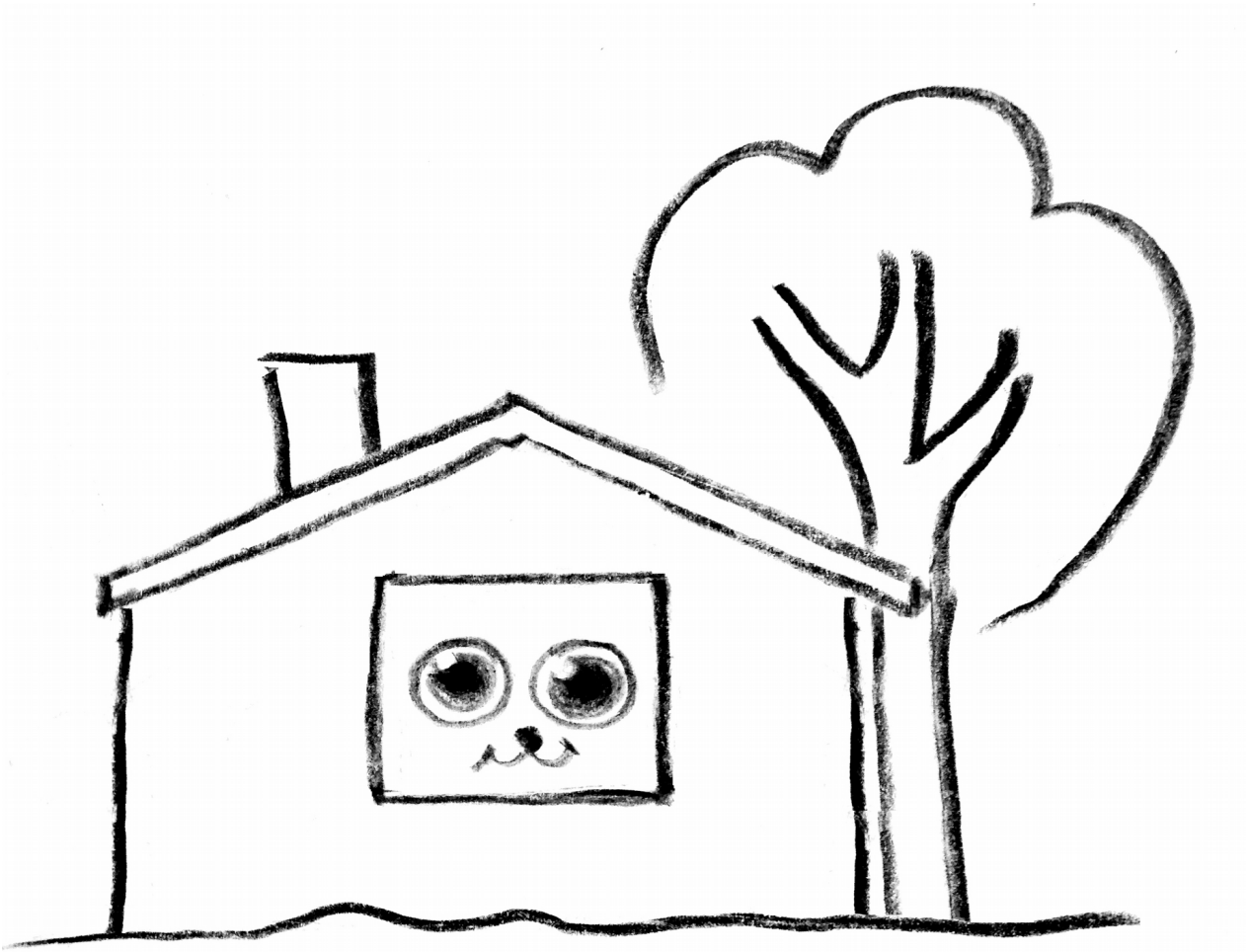
Micro-level

Docker ???



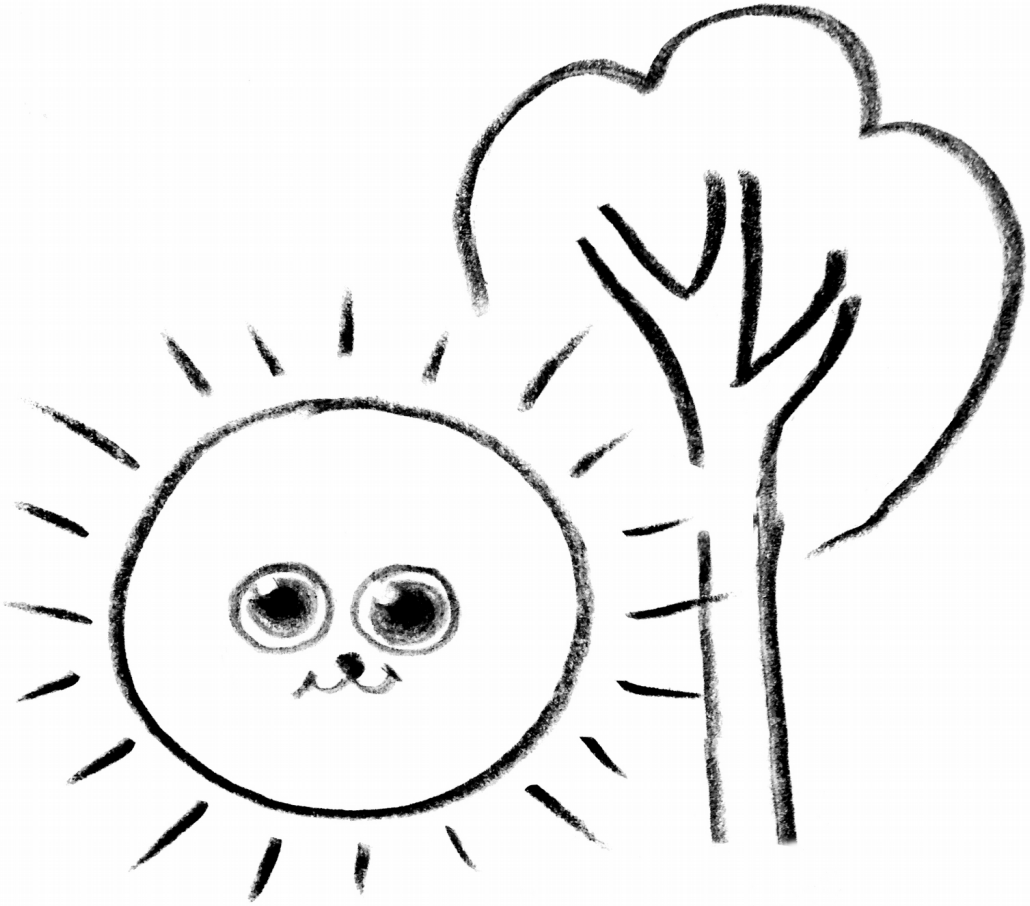
LINUX PITER

A service in a Docker-container



LINUX PITER

You'd imagine inside ...



LINUX PITER

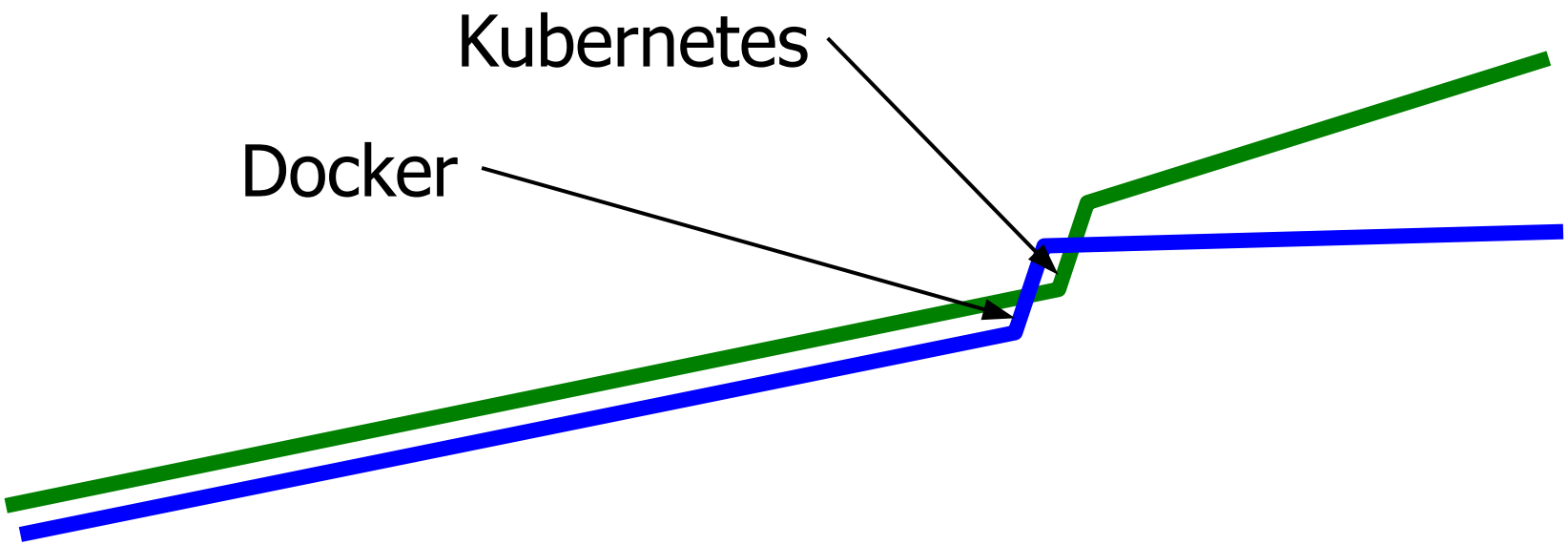
But in fact

LINUX PITER

The evolution of systems management

Distributed system, macro-level

Service application, micro-level



LINUX PITER

Docker for micro-level management

Simple provisioning

Dockerfile: **shell, deb/rpm, chef/puppet/salt**

Quick update: **layers**

Interface to macro-level: **manual**

Components integrations

Microserices: **separate containers, manual**

LINUX PITER

Micro-level, service application

Frequent updates

Many dependencies (components)

- Programs, libraries, date

Dependency versions

Components integration

Settings

Interface to the macro-level

LINUX PITER

Actual problems

Versions incompatibility

bugs or API changes in the new versions

bugs or vulnerabilities in the old versions

Human error

when updating

internal settings

orchestration meta-data

LINUX PITER

Easy rider

- Software development, corporate customers
- Customized software solutions
  - Business Intelligence
- Continuous integration
- Several hundred projects
  - Common components
  - Customized compoments
- Some merges & acquires before

LINUX PITER

Easy rider

Primary software

- Oracle, SAP Hana, PostgreSQL, MySQL
- Java, C++, Python, JavaScript, PHP

And

- R, Excel, ImageMagick, other stuff

OC

- RHEL, Oracle Linux, SLES
  - own patches for JRE, libc, OMP etc
- Windows

LINUX PITER

Easy rider

Delivery to the client

- SaaS
- Virtual machines
- .exe + jar
- RPM packages

LINUX PITER

Easy rider

Before

- Platform
  - X build servers
  - Different OSes
- Management
  - Scripts — make, shell, python
  - Jenkins
  - Puppet

LINUX PITER

Easy rider

Problems

- Poor testing
  - Long wait for the result → rarely tested
- Nonconformant test and the work environment
  - Errors raised on the customer side
- Build process often breaks
- A lot of manual labor

Easy rider

What is needed

- Well controlled environment
  - SaaS, Virtual machines
- Quality building and testing
  - Reproducible builds
  - Fast builds
  - Minimum manual intervention
  - Similar to Maven or Gradle

LINUX PITER

Easy rider

Considering

- Juju Charms
- Nix

Nix package manager

Built package — pure function of

- Source code
- Configuration values
- Dependencies (another built packages)
- Building rules

LINUX PITER

Nix

Everything consist of a packages

Every package

- Separate directory
- Read-only
- Is built when required as dependency
  - or installed pre-built from repository

LINUX PITER

Nix – user environment

- Union point for several **nix**-packages
- Every user environment — in a separate read-only directory
- Every packages combination — independent environment
- Adding or removing a package
  - Copy current environment to the new one
  - Add or remove package to the new environment
  - Switch current environment to the new (symlink replaced)

LINUX PITER

Nix – user environment

Directories structure complies FHS

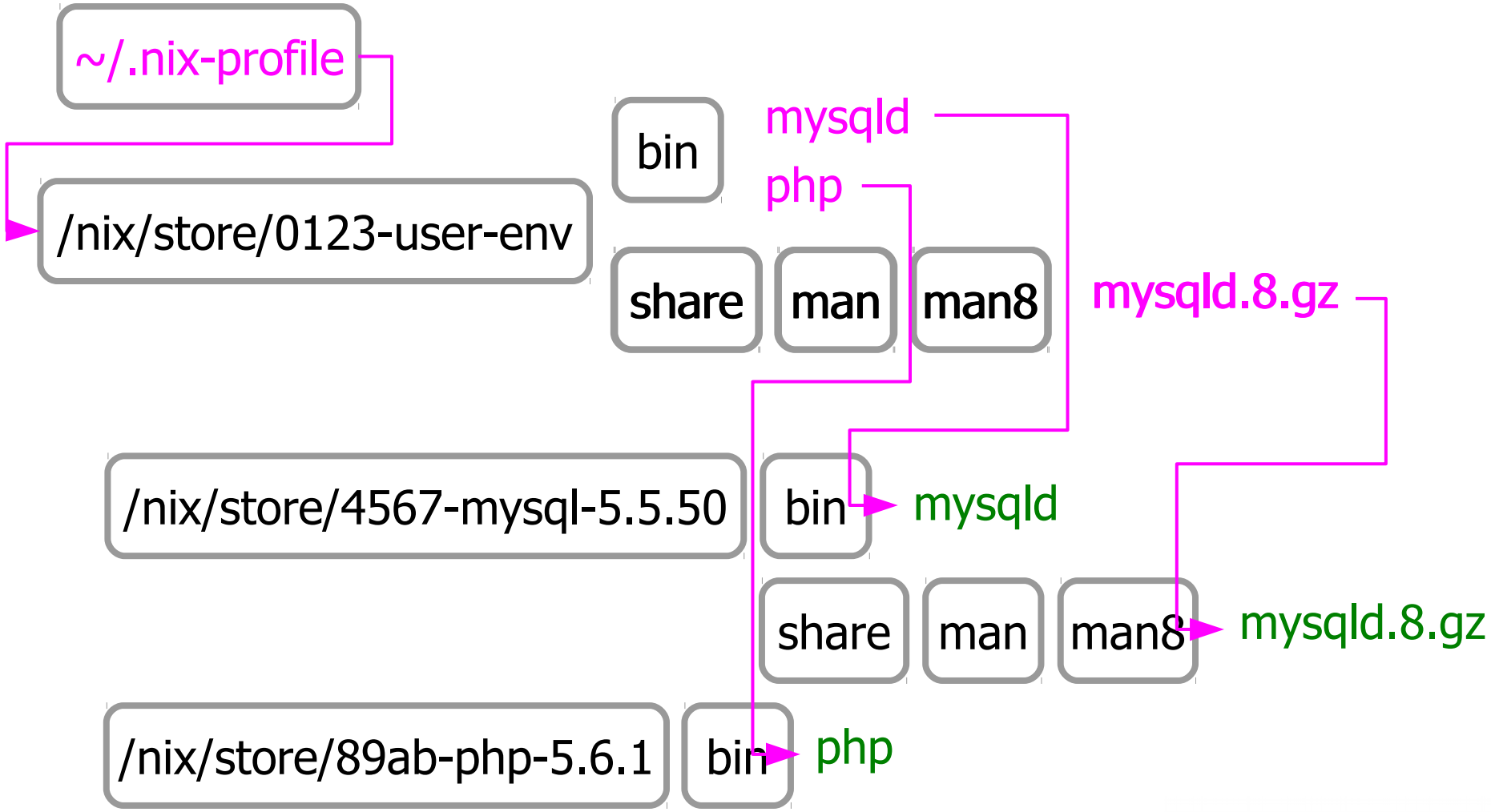
Present bin, etc, lib, include, share, libexec

Absent var, log, tmp — read-only

Symlinks from user environment to the corresponding files in a packages

LINUX PITER

Nix — user environment



LINUX PITER

Easy rider

Now

Hardware (SaaS & CI)

- ~ 180 own machines (SaaS — 90 & CI — 90)
- < 200 AWS (mostly CI)
- 500 — 2000 containers

Management

- Nix + Docker + Kubernetes
- Jenkins
- Puppet

LINUX PITER

Easy rider

Problems with Nix

- No version dependencies
- All dependency consumers rebuilding
  - Long wait for the result - up to 2 hours

Nix — rebuild on argument changes

`mysql: libc openssl`

`0123-libc 4567-openssl → 89ab-mysql`

patched **libc** → **`cdef-libc`**

`cdef-libc 4567-openssl → 3210-mysql`

Easy rider

Searching

- Has Nix's advantages
  - Automated builds
  - Dependencies
  - Isolation
  - Repeatability
- Plus
  - No unnecessary builds
  - Suitable for multiple versions

LINUX PITER

Easy rider

Considering

- Habitat
- Snappy
- Gentoo
- Jetware

Jetware vs Nix, in common

- Consists of a packages
  - Read-only
  - With versions
  - With build variations (flavours)
- Packages with dependencies
  - Automated installation
  - Automated building
- Configuration variables of a package

LINUX PITER

Jetware — packages

Not a pure function

Dependencies

on package, on version conditions

Incapsulated packages

Packages self-integration and self-configuration

LINUX PITER

Jetware — runtime environment

- Union point for packages
- Keeping packages data and settings
- Separated from OS environment
  - works directly on any OS
    - **libc** — just a package

LINUX PITER

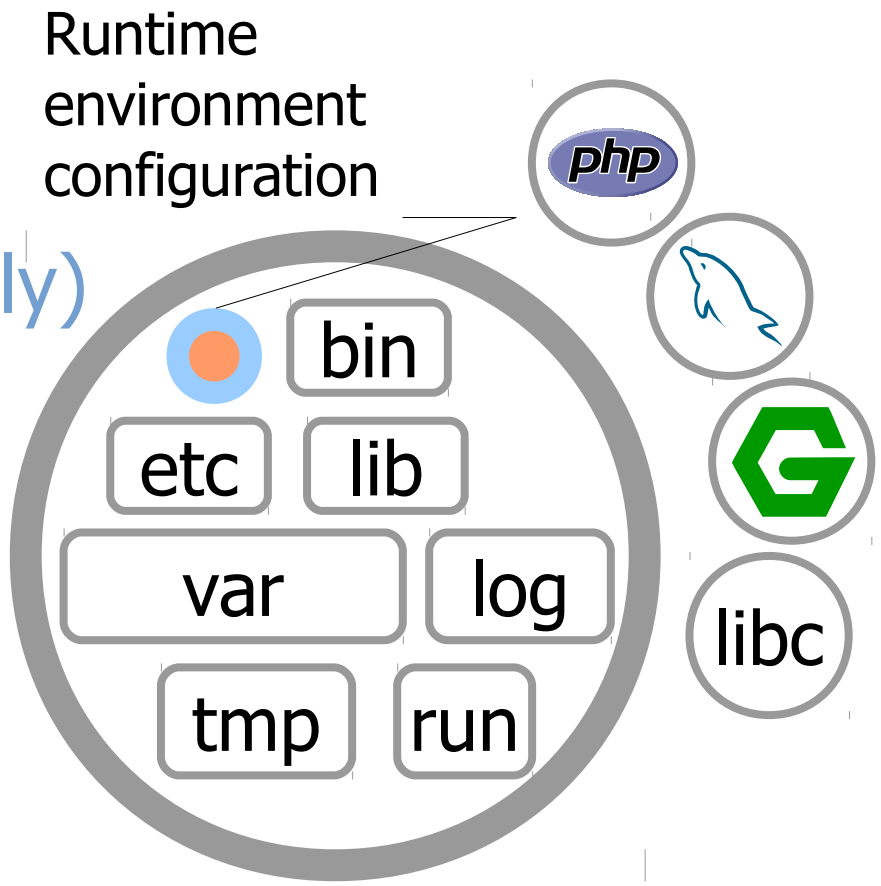
Jetware — runtime environment

Package — outside

Package — namespace

Package keeps all (read-only)

Data and settings — inside

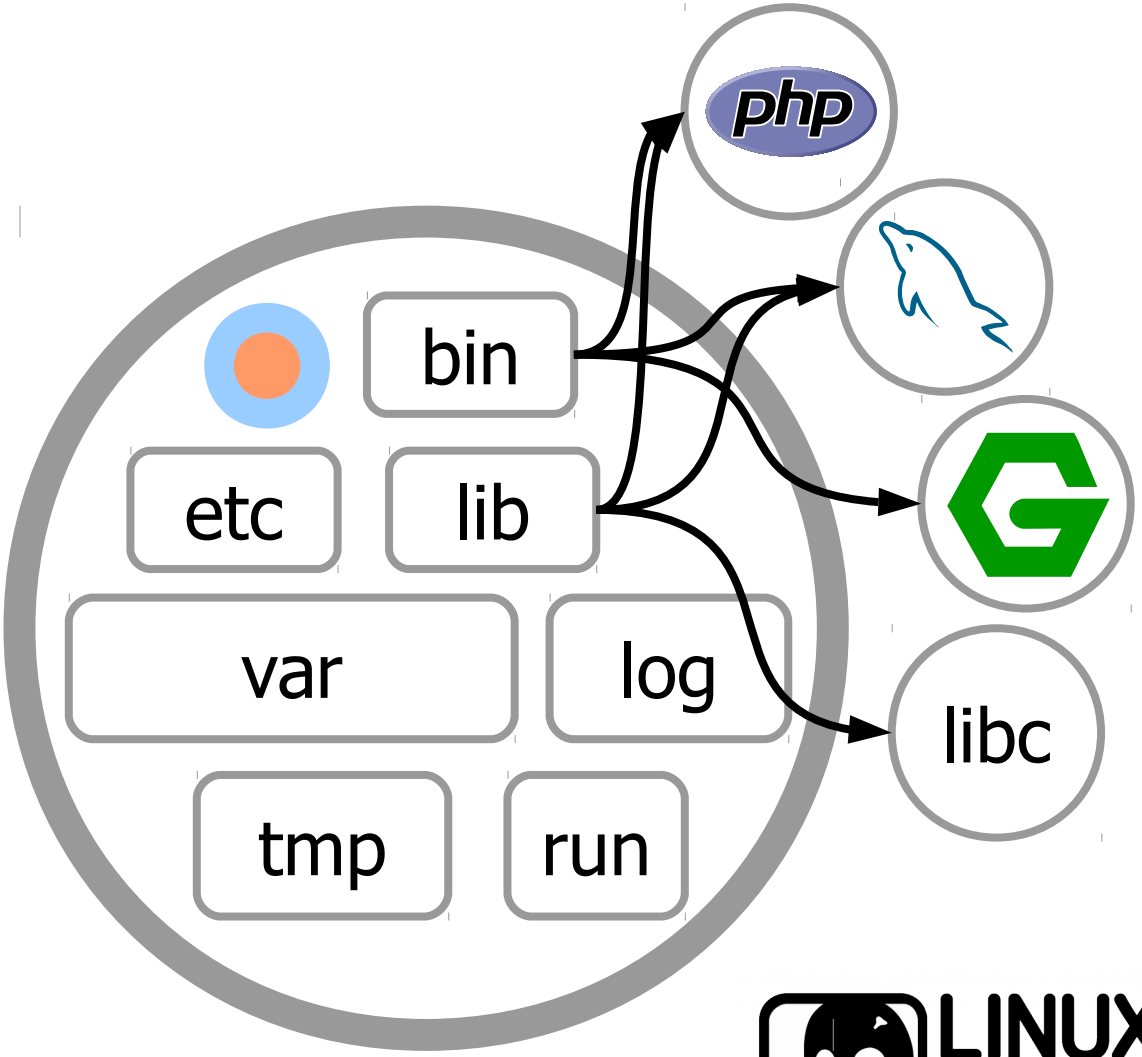


LINUX PITER

Jetware — export into environment

Immutable data

- Only minimum
- **bin, lib**
- symlinks

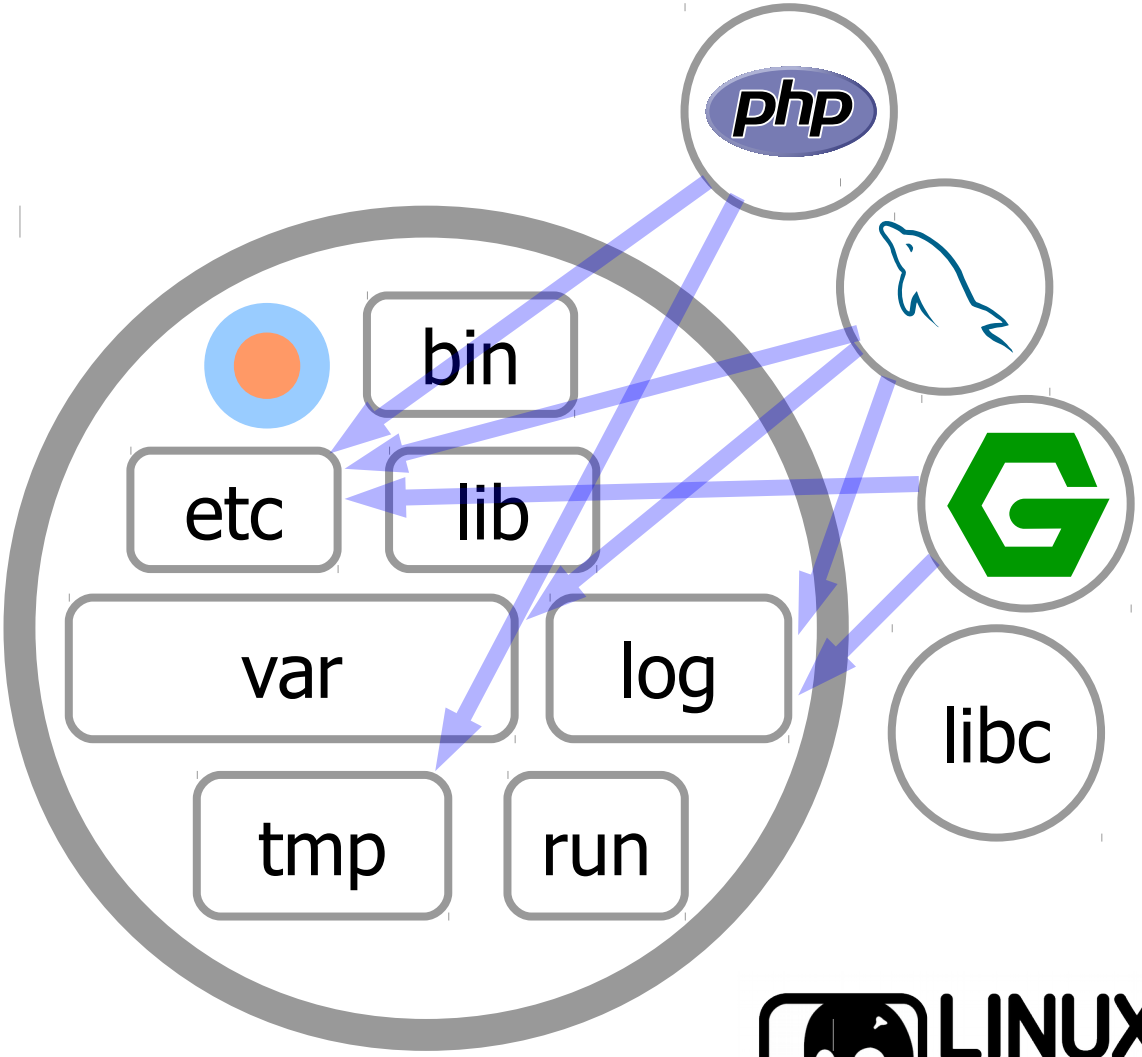


LINUX PITER

Jetware — export into environment

Changeable data

- Copied
- Separate sections
- Own FHS



LINUX PITER

Packages self-integration

Data export to another packages

Relations, files or API

Example

package **mysql** exports own C-headers into **cc** package

Settings value for another packages

Example

mongodb_tuning assigns `mongodb.port=22333`

Easy rider

Demo implementation with Jetware

- Builds by dependencies, versions
  - Own subprojects and third-party components
- Repositories with testing → staging flow
- Docker-container creation
  - 40 — 500 packages
  - 1.5 — 20 seconds

Expected

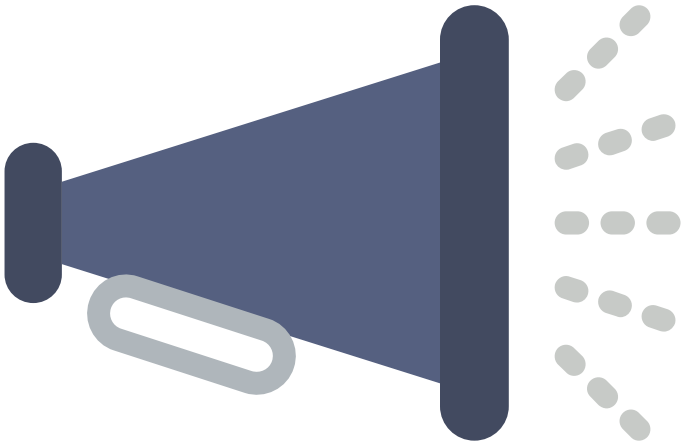
- ~ ~~180~~ 120 own machines (CI — 30, 90 — SaaS)
- < ~~200~~ 0 machines from AWS

LINUX PITER

The evolution of distributed systems management

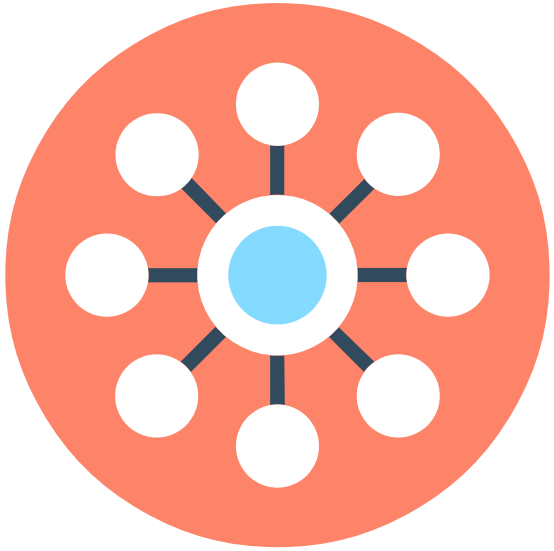
Macro-level

Orchestration



Micro-level

Self-constuction



LINUX PITER

What’s next?

Systems management fusion both on macro-level and micro-level

~ 1 — 3 years

LINUX PITER

Life cycle of a service

Developer

1 Source code + libraries

2 Compiling, building

3 Dependencies integration

4 Testing

5 Export to image

6 Deployment

7 Put into operation

???

Orchestration

LINUX PITER

Does the program need OS?

Program needs

- Kernel for syscalls
- File system
- Runtime environment
  - Libraries and executables
  - Another data — resources

OS environment — not needed

LINUX PITER

Must the program be implanted to OS?

To access the program we need

- To know where it is placed
- To feed the data input
- To accept the data output

The program can be anywhere — OS is not needed

LINUX PITER

Tight binding of OS and application

Historical heritage

Influence of Linux distributions

Self-constructing autonomous program

Is not bound to OS

Describes

- Runtime dependencies (packages, services)
- Interface
  - Consuming and suppling resources (port, stdin/stdout etc)
- Build rules (optional)
  - Sources
  - Build dependencies

LINUX PITER

Management on macro-level

Scheduling object

- Application: package and version

Starting an application

- Package building or retriving from a repository
- Runtime environment construction
- Service deployment and execution
- Publishing interface data for service discovery

LINUX PITER

Service package description

```
postgresql-cluster-user-geotags-5.1
 requires:
  postgresql-9.1
  postgresql-cluster-config-2.1
  storage-ssd-1:
   env:
    volume: user_geotags
    size: 1gb
```

LINUX PITER

Versions and updates

Developer releases the version

- commit in repository tagged with version

Another service has reference on package-version

- service is building and executing

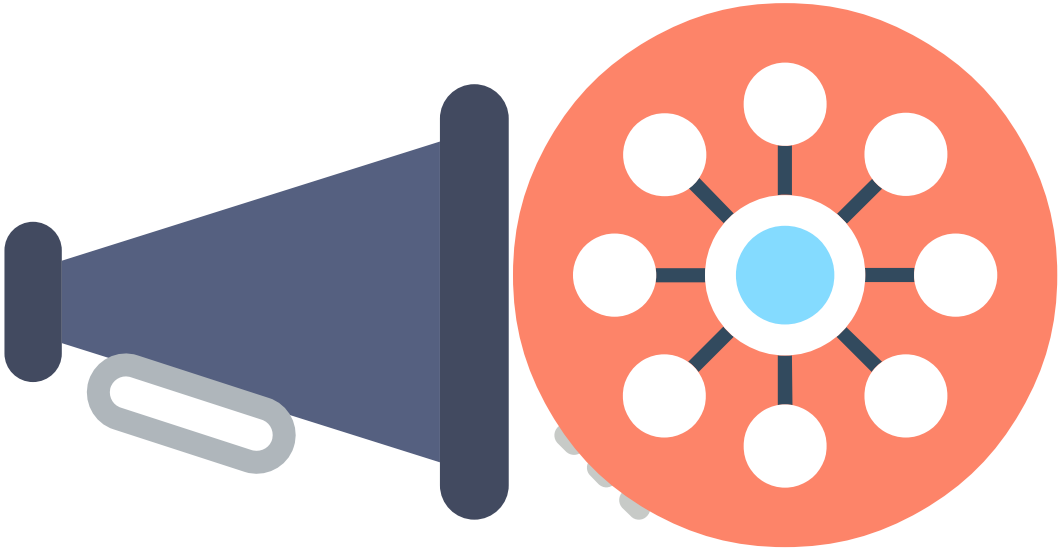
No reference on package-version

- garbage collector stops the service

LINUX PITER

The evolution of distributed systems management

Macro-level and micro-level fuse



LINUX PITER

Robots replacing devops

Developers and sysadmins returning to the more intellectual work

And then?

Some years later

The fifth stage of evolution - self-organization

LINUX PITER

How devops exhausts itself, and what will happen next

Kirill Vechera

http://jetware.org
Skype: kirill.vechera
cv-c@jetware.org

LINUX PITER