Overview of A Pattern Language for Engineering Software for the Cloud

Tiago Boldt Sousa, tbs@fe.up.pt, Department of Informatics Engineering, Faculty of Engineering, University of Porto
Hugo Sereno Ferreira, hugosf@fe.up.pt, Department of Informatics Engineering, Faculty of Engineering, University of Porto
Filipe Figueiredo Correia, filipe.correia@fe.up.pt, Department of Informatics Engineering, Faculty of Engineering, University of Porto

Software businesses are continuously increasing their cloud presence in the cloud. While cloud computing is not a new research topic, designing software for the cloud still requires engineers to make an investment to become proficient working with it.
This paper introduces a pattern language for cloud software development and briefly describes details pattern. Design patterns can help developers validate or design their cloud software. The language is composed by ten patterns novel patterns organizes in three categories: Orchestration and Supervision, Monitoring and Discovery and Communication.
Finally, the paper demonstrates how to adopt the pattern language using a pattern application sequence.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**] Patterns

General Terms: Design

Additional Key Words and Phrases: Cloud Computing, Design Patterns, Software Engineering

ACM Reference Format:

Sousa, T.B., Ferreira, H.S., Correia, F.F.. 2018. Overview of a Pattern Language for Engineering Software for the Cloud. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 54 pages.

1. INTRODUCTION

Today, over 49% of the world population has access to at least one Internet enabled device [Internetlivestats.com 2016]. Exploiting this scale with cloud applications made the Internet an appealing channel for running businesses. Such resulted in an explosion in the adoption of public cloud services, with it's market surpassing US$204 billions in 2016 [Woods and Meulen 2016].

Motivated by such widespread of the Internet and the explosive growth of software businesses built on top of it, software engineering became one of the fastest expanding branches of engineering. In fact, the demand for new engineers grows at an higher rate than the pace at which they are graduating [Taft 2015].

Using the cloud as a foundation for application development introduces new challenges and is essential for any Internet business. Still, there is a clear lack of research supporting developers design decisions while for cloud software, namely, identifying what forces drive successful cloud software, and the guidelines to optimize them in order to craft better cloud software.

With this research, the author proposes a pattern language for building cloud software, helping development teams make informed architectural decisions that will improve their software operations. The pattern language can be used by less experienced teams to bootstrap their design decisions or by experts to validate their own existing architectures. Finally, it provides a framework for reasoning, given that the concepts introduced by the pattern language can be used to argue about cloud architectures using the ontology it introduces.

2. A PATTERN LANGUAGE FOR ENGINEERING SOFTWARE FOR THE CLOUD

This research contributes to the field of software engineering with a pattern language that helps in the design process of cloud software architectures. This section introduces the research questions of this work, presents the pattern language structure and briefly describing each pattern and, finally, demonstrates how it can be applied using an example scenario. The patterns that compose the pattern language are described in Appendix A.

2.1 Mining the Patterns

While capturing this pattern language, some considerations where taken in order to ensure the individual quality of each pattern. Inspired by the evaluation framework proposed by Seidel [Seidel 2017], the following attributes were considered:

Completness: Is the pattern description complete? [Alexander 1979] A complete pattern provides a level of detail that enables the reader to identify with the problem and reproduce the solution.

Briefness: Does the pattern contain more information than what's strictly needed? A brief pattern goes straight to the point, being easy to read and reproduce.

Validity: Is the stated solution valid and with enough known uses described? A valid pattern documents an accepted good solution and justifies it with accurate examples.

2.2 How To Read These Patterns

The patterns described in this work have all been documented using the pattern structure described below, strongly inspired by classical pattern templates [Meszaros 1998; Wellhausen and Fiesser 2011]. Each pattern is composed by the following sections:

Abstract: A brief description of the pattern and its applications.

Context: The circumstances that result in the manifestation of the problem. By reading this section, the reader would be able to understand what is the driver for the problem. Experienced users will often relate the context with their previous experiences.

Example: Describes a concrete scenario aligned with the context, where the problem is observable, highlighting the intricacies that makes it a complex problem to solve.

Problem: Formalizes the problem, detailing on why it is complex to be solved.

Forces: Identifies the forces that influence the design of the solution. Forces commonly oppose each other, leaving for the reader to decide how to properly balance them to customize the pattern's implementation to his specific needs.

Solution: Describes how the pattern addresses the problem and describes its implementation details, which often need to be adapted considering how the forces are balanced.

Example Resolved: Describe how the pattern can be instantiated in order to address the scenario described in the example above.

Resulting Context: Elaborates on the benefits and liabilities introduced by the implementation of this pattern.

Related Patterns: Identifies which patterns can be used with or are incompatible with the implementation of this pattern.

Known Uses: Pattern should be extracted from recurring solutions to the same problem observed in the wild. The section identifies implementations that motivated the writing of this pattern.

Further Considerations: An optional section in the pattern, where additional details are shared or a discussion is held, elaborating on the intricacies of the pattern.

This structure is a superset of *A Pattern Language for Pattern Writing* [Meszaros 1998], which suggested the common structure of context, problem, forces and solution and it is so structures in order to facilitate the reader's navigations through the pattern, easily identifying or skipping the information he is looking for or is not interested in, respectively.

2.3 Pattern Language Overview

This pattern language has been organized into four pattern categories. This subsection describes each of those categories as well as it briefly describes the patterns in the category. The full version of each pattern can be found in Appendix A.

2.3.1 *Automated Infrastructure Management.*

The patterns referenced in this section are not introduced by this pattern language, but a reference to them is essential, as they provide the grounds to much of what this pattern language builds upon. Automated Infrastructure Management if often associated with DevOps and has been extensively researched and has such will remain outside the scope of this research [Cycligent ; Loukides 2012; Reed 2014; XebiaLabs ; Technology 2014; Erich et al. 2014].

INFRASTRUCTURE AS CODE is one of the most relevant contributions from the DevOps mindset to software engineering for the cloud. This pattern handles the management of infrastructure using software [Smeds et al. 2015; Hüttermann 2012; Riley 2015; Amazon 2018]. For that, the creation and changes to the infrastructure are defined during the development process, following the development practices in place by the team. This will usually ensure the quality of the software created to manage the infrastructure through practices such has peer review. Furthermore, INFRASTRUCTURE AS CODE, following the DevOps mindset, deprecates the need for knowledge operation workers that would be dedicated to setup and handle infrastructure. The team owns that responsibility, managing it as part of the development cycle. This automation will further increase confidence in changes, as it eliminates the possibility of human introduced error.

Another known pattern required by this pattern language, this time available from cloud providers, is AUTOMATED SCALABILITY. This pattern enables a cloud provider to monitor a set of instances for their resource allocation, scaling them appropriately to balance performance and costs [Zhang et al. 2010; Cycligent ; Fehling et al. 2014]. As an example of how this pattern works, an infrastructure with ten instances that is experiencing increased CPU usage can be automatically scaled, one machine at the time, until the average CPU load in the infrastructure is below a configured maximum threshold. On the other hand, if the average CPU usage is below a minimum threshold, the infrastructure size can be reduced, one machine at the time, until the average CPU load it above a desired average threshold. This will ensure that the infrastructure is maintains an overall CPU usage within the desired threshold interval.

2.3.2 *Orchestration and Supervision.*

Infrastructure empowering software in the cloud is typically volatile and dynamically allocated. As such, orchestration plays a key role at dynamically identifying the execution setup and adapt the software to cope with it. The patterns in this section describe how to setup the necessary hardware and software to orchestrate services in the cloud and insure they run flawlessly.

Creating development or production environments manually is a time consuming process. The probability of error is high, given the commonly large number of dependencies and configurations required. Furthermore, these get scattered in the host making managing the machine and hosting multiple applications troublesome. While virtualization can be used to create a portable environment of the entire hardware and software stack, it always virtualizes the whole hardware and software stack, which is very resource demanding. CONTAINERIZATION is a better alternative, enabling the creation of immutable, reproducible, portable and secure software execution environments. Containers are considerably more lightweight than full stack virtualization, as there is no need to virtualize the operative system layer. Containers avoid the need to install dependencies and have configurations scattered within the host, making them easier to manage and deploy at scale [Boldt Sousa et al. 2015; Scheepers 2014]. Having at most one service per container ensures that all services are isolated from each-other, preventing

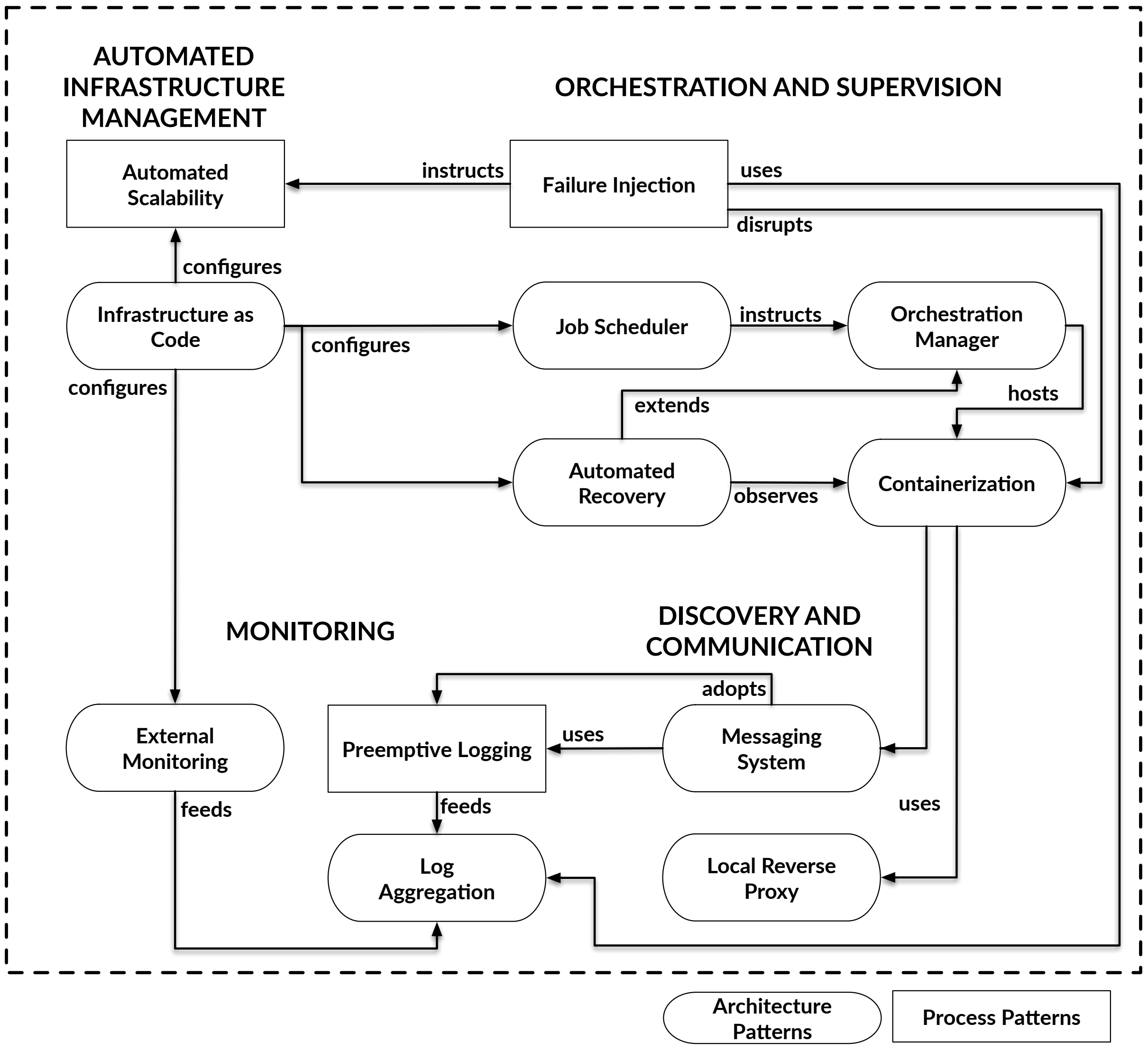


Fig. 1: Pattern map for engineering software for the cloud. Patterns are identified as architecture or process patterns and aggregated into one of the four categories. Arrows identify the relationships between the patterns.

secondary effects in their behaviors. This approach is also essential for individually scaling each microservice. Adopting containers facilitates the service's portability. Configurations should not exist inside the container, but instead services should acquire their configuration from environment variables made available to the container in each specific execution environments. Environment-based configuration enables not only the service's configuration to be injected into the environment, but the host resources' details to be made available in the same way.

Servers in an infrastructure might differ in hardware. While some might provide more CPU, others might have higher amounts of RAM available. Likewise, services requirements also are unique. While some might require a specific amount of memory to be available, other might have the need to be co-located in the same host for latency purposes. As such, services need to be co-located within hardware fulfills their resource and logical requirements. An ORCHESTRATION MANAGER should be responsible for allocating services to the proper hosts, considering their overall and available resources, as well as any other allocation restrictions that the service might have [Boldt Sousa et al. 2015; Odewahn 2014; Hausenblas]. This pattern works best when services are distributed using the already described CONTAINERIZATION pattern.

Asynchronous tasks, such as database maintenance, sending emails or performing backups, are common in cloud software. These might run at a given frequency or at a single point in time. JOB SCHEDULER can be used to orchestrate the execution of these tasks in an infrastructure and evaluate their result, generating error reports when need [Boldt Sousa et al. 2018b].

Software fails. That assumption is even further relevant while orchestrating software in the cloud, given its typically large scale. Accepting that it is not possible to prevent software from failing, supervision ensures that services are running as expected, executing the proper action to recover them in case of failure.

Services running inside containers should be resilient in case of failure, providing AUTOMATED RECOVERY. Exploiting the immutability of containers, the container shall restart itself automatically to try to recover the service whenever it detects a malfunction. Advanced strategies might be applied to recover a service, or set of services, such as restarting a list of services in a specific order. The ORCHESTRATION MANAGER should decide on the best strategy for each scenario [Boldt Sousa et al. 2018b].

Cloud Software has typically a desired uptime of 100%. Redundancy is often a first step to guarantee availability. Still, it is a fact that software fails [Charette 2005]. As such, developers need to ensure that their application is resilient to failures and able to recover to a working status without human intervention, with mechanisms for improving resilience being built into the cloud application. To increase confidence in these mechanisms they need to be themselves tested. And since the environment might influence how the system behaves, the production system's resilience itself must be tested. To do so a FAILURE INJECTION mechanism can periodically inject unexpected events in the system, evaluating if it continues to behave appropriately [Boldt Sousa et al. 2018a]. Such events can range from sending invalid inputs to the application to shutting down one of the servers hosting the application. In both scenarios, if there is any unexpected impact into the application, the resilience mechanisms should be triggered and the application have little to no impact in its availability. Whenever that doesn't happen, the developers can work on improving the resilience behavior. FAILURE INJECTION is commonly executed under close supervision and in redundant systems to prevent actual damage to the application.

In summary, the following patterns are introduced in this section:

Orchestration Manager: This pattern deals with service allocation in an infrastructure when each server can host multiple isolated applications. It takes into consideration the resources available in each machine in the infrastructure and the service's requirements, ensuring an optimal placement and execution.

Job Scheduler: A JOB SCHEDULER enables the programed execution of single time and recurrent tasks in an infrastructure. In the context of the cloud the scheduler cooperates with an ORCHESTRATION MANAGER in order to leverage the available infrastructure and its service placement to optimally execute the programmed tasks.

Automated Recovery: Configure health checks along with service definition to periodically evaluate its health, attempting an automated recovery of unhealthy containers by restarting them.

Failure Injection: FAILURE INJECTION is responsible for generating atypical events at both the application and infrastructure level, exercising the implemented recovery mechanisms in such way that the resilience of the application is continuously verified.

Containerization: This pattern addresses the need for a portable, efficient and secure environment for executing applications. An issue that is most troublesome when deploying multiple applications in the same host, which a) can result in extremely complex dependency management scenarios and clutter the system or b) requires virtualization which requires part of the host's resources.

2.3.3 *Monitoring.*

Monitoring continuously evaluates the status of the services composing a cloud application, reporting back to the team when an issue is identified. It works reactively by detecting issues on data produced by the services, using log entries or alarms as inputs, or proactively by interacting with the services to verify their status.

While designing the application, all public endpoints and their expected behavior should be identified, so that an EXTERNAL MONITORING tool can be configured to monitor them. This enables the unbiased observation of the application's state from outside its infrastructure, ensuring that the monitoring tool observes the same as an user would [Boldt Sousa et al. 2018a].

Debugging an application in production requires as much information as possible in order to trace the actions that lead to an issue. Services should LOG AGGREGATION, producing verbose execution logs that should be kept for the longest period of time possible [Boldt Sousa et al. 2017].

Having distributed services producing logs will required developers to leverage multiple log files to trace an issue. To prevent this, the team should adopt LOG AGGREGATION, by having a centralized view of all the logs generated by all services in a queryable format [Boldt Sousa et al. 2017].

The following patterns are introduced in this section:

External Monitoring: Monitor the application's public interfaces from an external source, imitating the user's experience, and notify the team about unexpected behavior.

Preemptive Logging: Adjust log verbosity preemptively, making relevant execution information available for later debugging, statistics, and event playback.

Log Aggregation: Aggregate and index all service and server logs, providing the team with a single point to query and visualize information from the logs.

2.3.4 *Discovery and Communication.*

Most cloud applications are composed by multiple services cooperating towards providing the whole application in what it is typically called a microservice architecture [Lewis and Fowler 2014]. Being deployed in containers hosted in dynamically provisioned hardware, the services must first discover and create a communication channel before they can start to cooperate.

While using an ORCHESTRATION MANAGER that dynamically allocates containers, the exact network location at where a service will be running is unknown. Using a LOCAL REVERSE PROXY, a service can be abstracted through a local network port exposed on every machine that is always forwarded to one instance of the service, possibly balancing traffic between multiple instances [Schumacher et al. 2006; Boldt Sousa et al. 2015]. This is easily achieve by preemptively creating a table that maps local ports to services. Whenever the port is mapped, the service is up and the communication can be established.

Some use cases require services to communicate amongst themselves synchronously for RPC and asynchronously for delegating information to collaborating services. A MESSAGING SYSTEM can be used to send both types of messages between micro-services, eliminating the complexity associated with service discovery [Gawlick 2002; Boldt Sousa et al. 2017].

This section introduced the following sections:

Messaging System: Use a messaging mechanism, colloquially known as message queue, to abstract service placement and orchestrate messages with multiple routing strategies between them.

Local Reverse Proxy: This pattern deals with service discovery when deploying services that should cooperate in dynamically provisioned hardware, who lack each-others' network addresses.

3. ADOPTING THE PATTERN LANGUAGE

Resistance to change is by itself a pattern [Dent and Goldberg 1999]. Adopting a pattern language for developing software for the Cloud requires teams to adapt their mindset regarding their organization, processes and software architectures. While it is imperative that the team is motivated to change, this pattern language eases its adoption as it can be partially implemented. The team can identify its most critical problem and implement the pattern or set of patterns that solve it without addressing the whole pattern language. The next section demonstrates how a development team could gradually adopt the described patterns in a sequence of iterations.

Amongst the more experienced users, the pattern language can be used as a ontology for reasoning about cloud architectures, as well as a validation tool for their already existing architectures. Using it as reference will also facilitate how experts share their ideas with less experienced engineers.

This section was inspired by *The Unfolding of a Japanese Tea Garden* by Christopher Alexander [Alexander 2006]. It uses a sequence as a way to help the reader understand how the patterns relate and complement each other. Sequences describe a set of actions that should follow each other in order to achieve a specific goal.

Consider the scenario where a cloud *practitioner* needs to create and deploy a redundant Web Application, composed by a client-facing HTTP server and a database.

The *practitioner* should design his HTTP server and database as two cooperating microservices. By using CONTAINERIZATION and one service per container, he would create two container images, one of each service. These containers would be highly portable between multiple environments such as local, staging or production environments, configured using the available environment variables.

Using INFRASTRUCTURE AS CODE the *practitioner* would describe the initial infrastructure required to setup the system. By executing this programmatic description, the required infrastructure would become available. AUTOMATED SCALABILITY could be setup to ensure that the hardware where the web server executes would scale horizontally if needed according to the provided scalability rules.

To deploy his services in an isolated and scalable way, the infrastructure would be abstracted thought an ORCHESTRATION MANAGER, which would be responsible for allocating the containers machines in the infrastructure optimally, taking into consideration the total and available resources in each machine.

JOB SCHEDULER would be responsible for executing the daily database backup process to an external site.

To facilitate discovery in the dynamically allocated hardware, the web server would use the local network port 12345 to connect to the database. Such would be possible given a LOCAL REVERSE PROXY configured in all machines that would expose a static service port for each service instantiated. This scenario would not require the MESSAGING SYSTEM.

To ensure the service is working properly, the *practitioner* would implement the following monitoring techniques.

An EXTERNAL MONITORING service can monitor all public application endpoints, ensuring that they are both online and responding appropriately.

To further increase awareness over the system's state, PREEMPTIVE LOGGING could be adopted to configure the developed and adopted services to use an appropriate level of logging to make the required historical information available to the team to develop issues after they have happened. LOG AGGREGATION can bring all this information to a centralized, indexed and queryable location for easier use.

Finally, and in order to validate the resilience in the system, FAILURE INJECTION can exercise the resilience mechanisms by randomly introducing errors in the infrastructure, such as randomly shutting down machines, and verifying that the system recovers automatically.

4. SUMMARY AND FUTURE WORK

This paper describes a pattern language for engineering software for the cloud. The language is divided into four categories: Automated Infrastructure Management, which references known DevOps patterns upon which this work build; Orchestration and Supervision, proving strategies to abstract an infrastructure's resources and automate service placement within it; Monitoring, which helps the team be aware of the system's status and Discovery and Communication which facilitates inter-service communication within the infrastructure.

This language aims practitioners to make informed decisions while designing their cloud environments, facilitating the creation of resilient and reliable cloud applications.

Plans for future work are twofold. First, continue increasing the completeness of this pattern language by increasing the number of patterns the compose it. Second, increase the patterns quality by surveying experts to gather their input on each individual pattern's attributes: completeness, briefness and validity.

5. ACKNOWLEDGEMENTS

The authors would like to acknowledge Robert Hanmer for his feedback while shepherding this paper.

REFERENCES

Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. 2013. Cloud monitoring: A survey. *Computer Networks* 57, 9 (2013), 2093–2115. DOI:http://dx.doi.org/10.1016/j.comnet.2013.04.001

Christopher Alexander. 1979. The Timeless Way of Building. *New York Oxford University Press* (1979).

Christopher Alexander. 2006. *The Nature of Order: The Process of Creating Life*. Center for Environmental Structure. 636 pages.

Amazon. 2015. Amazon EC2 Container Service. (2015). https://aws.amazon.com/docker/

Amazon. 2017a. https://aws.amazon.com/cloudtrail/. (2017). https://aws.amazon.com/cloudtrail/

Amazon. 2017b. Scheduled Tasks (cron). (2017). http://docs.aws.amazon.com/AmazonECS/latest/developerguide/scheduled_tasks.html

Amazon. 2018. AWS Auto Scaling. (2018). https://aws.amazon.com/autoscaling/

Arcitura Education Inc. Dynamic Failure Detection and Recovery. (????). http://cloudpatterns.org/design_patterns/dynamic_failure_detection_and_recovery

Arcitura Education Inc. 2017. Cloud Patterns. (2017). http://cloudpatterns.org/

Azure. 2017. Azure Logging and Auditing. (2017). https://docs.microsoft.com/en-us/azure/security/azure-log-audit

Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. 2015. Patterns for Software Orchestration on the Cloud. In *Proceedings of the 2015 Conference on Pattern Languages of Programs*.

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2017. Engineering Software for the Cloud: Messaging Systems and Logging. *Proceedings of the 22Nd European Conference on Pattern Languages of Programs* (2017). DOI:http://dx.doi.org/10.1145/3147704.3147720

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018a. Engineering Software for the Cloud : External Monitoring and Fault Injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. 1–13.

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018b. Engineering Software for the Cloud: Automated Recovery and Scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. 1–13.

Jonas Bonér, Martin Thompson, Dave Farley, and Roland Kuhn. 2014. The Reactive Manifesto v2.0. (2014). http://www.reactivemanifesto.org

Thanh Bui. 2015. Analysis of Docker Security. *Computing Research Repository* abs/1501.0 (2015), 7. http://arxiv.org/abs/1501.02967

F Bushmann, R Meunier, and H Rohnert. 1996. Pattern-oriented software architecture: A System of Patterns, Volume 1. *John Wiley&Sons* 1 (1996), 476. DOI:http://dx.doi.org/10.1192/bjp.108.452.101

James Casey, Lionel Cons, Wojciech Lapka, Massimo Paladin, and Konstantin Skaburskas. 2011. A Messaging Infrastructure for WLCG. *Journal of Physics: Conference Series* 331, Part 6: Grid and Cloud Middleware (2011). DOI:http://dx.doi.org/https://doi.org/10.1088/1742-6596/331/6/062015

Chaos Community. 2017. Principles of Chaos Engineering. (2017). http://principlesofchaos.org/

Robert N. Charette. 2005. Why Software Fails. (2005). http://spectrum.ieee.org/computing/software/why-software-fails

Chronos. 2017. Chronos. (2017). https://mesos.github.io/chronos/

CoreOS Community. 2015. CoreOS Project Page. (2015). https://coreos.com/

Ward Cunningham. 2014. Let It Crash. (2014). http://wiki.c2.com/?LetItCrash

Cycligent. *Continuous Delivery Patterns for Design & Deployment*. Technical Report.

DataDog. 2018. Docker Adoption. (2018).

Maximilien De Bayser, Leonardo G. Azevedo, and Renato Cerqueira. 2015. ResearchOps: The case for DevOps in scientific applications. *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015* (2015), 1398–1404. DOI:http://dx.doi.org/10.1109/INM.2015.7140503

Eric B Dent and Susan Galloway Goldberg. 1999. Challenging " Resistance to Change ". 35, 1 (1999), 25–41. DOI:http://dx.doi.org/10.1177/0021886399351003

Docker. 2018. Dockerfile reference. (2018). https://docs.docker.com/engine/reference/builder

Elastic. 2017. The Open Source Elastic Stack. (2017). https://www.elastic.co/products

Floris Erich, Chintan Amrit, and Maya Daneva. 2014. Report : DevOps Literature Review. (2014).

Thomas Erl, Robert Cope, and Amin Naserpour. 2015. *Cloud Computing Design Patterns*. 552 pages. informit.com/phLibrary

Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns*. 239–286 pages. DOI:http://dx.doi.org/10.1007/978-3-7091-1568-8

Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2014. An Updated Performance Comparison of Virtual Machines and Linux Containers. *Technology* 25482 (2014).

E.B. Fernandez. 2013. *Security patterns in practice: Building secure architectures using software patterns*. Wiley Series on Software Design Patterns.

Eduardo Fernandez-Buglioni. 2013. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. 582 pages.

Martin Fowler. 2017. What do you mean by "Event-Driven"? (2017). https://martinfowler.com/articles/201701-event-driven.html

Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014* (2014), 24–33. DOI:http://dx.doi.org/10.1145/2591062.2591175

Dieter Gawlick. 2002. Message Queuing for Business Integration. *{eAI} Journal* October 2002 (2002), 30–33.

Gitlab. 2017. Postmortem of database outage of January 31. (2017). https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/

Sébastien Goasguen. 2016. *Docker in the Cloud* (second edi ed.). O'Reilly Media.

Google. 2015. Google Cloud Container Service. (2015). https://cloud.google.com/container-engine/

Google. 2018. Reliable Task Scheduling on Google Compute Engine. (2018). https://cloud.google.com/solutions/reliable-task-scheduling-compute-engine

Robert Hanmer. 1998. An Input and Output Pattern Language. *Plop2* c (1998), 1–35.

Michael Hausenblas. *Docker-Networking-and-Service-Delivery*.

Peter Herrmann, Alexander Svae, Henrik Heggelund Svendsen, and Jan Olaf Blech. 2016. Collaborative Model-based Development of a Remote Train Monitoring System. In *Evaluation of Novel Approaches to Software Engineering, COLAFORM Track*.

Edward Hieatt and Rob Mee. Repository Pattern. (????). https://martinfowler.com/eaaCatalog/repository.html

Benjamin Hindman, Andy Konwinski, and Matei Zaharia. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *Nsdi* (2011). DOI:http://dx.doi.org/10.1109/TIM.2009.2038002

Robert Hof. 2016. Meet Project Storm, Facebook's SWAT team for disaster-proofing data centers. (2016).

Gregor Hohpe and Bobby Woolf. 2003. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. *Enterprise integration patterns designing building and deploying messaging solution* (2003), 736. DOI:http://dx.doi.org/10.1525/vs.2009.4.3.toc

Michael Hüttermann. 2012. Infrastructure as Code. In *DevOps for Developers*. Apress, Berkeley, CA, 135–156. DOI:http://dx.doi.org/10.1007/978-1-4302-4570-4{_}9

IEEE and The Group Open. 2016. crontab. (2016). http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html

Open Container Initiative. 2015. Open Containers Project Page. (2015). http://www.opencontainers.org/

Internetlivestats.com. 2016. Number of Internet users in the world. (2016). http://www.internetlivestats.com/internet-users/

Petros Koutoupis. 2018. Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers. (2018). https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-ii-working-linux-containers-lxc

Kubernetes. Run a Stateless Application Using a Deployment. (????). https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/

Kubernetes. 2017. Kubernetes Cron Jobs. (2017). https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/

Kubernetes. 2018. Pod Lifecycle. (2018). https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

Nelson G M Leme, Eliane Martins, and Cecília Rubira. 2001. A Software Fault Injection Pattern System (1) II . Architectural Pattern : Fault Injector. In *Pattern Languages of Programs*. https://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/ngmleme3/PLoP2001_ngmleme3_3.pdf

James Lewis and Martin Fowler. 2014. Microservices. (2014). http://martinfowler.com/articles/microservices.html

Mike Loukides. 2012. *What Is DevOps?* O'Reilly Media. 15 pages. http://shop.oreilly.com/product/0636920026822.do

L Magnoni. 2015. Modern Messaging for Distributed Sytems. *Journal of Physics: Conference Series* 608 (2015), 012038. DOI:http://dx.doi.org/10.1088/1742-6596/608/1/012038

Y.K. Malaiya, M.N. Li, J.M. Bieman, and R. Karcich. 2002. Software reliability growth with test coverage. *IEEE Transactions on Reliability* 51, 4 (2002), 420–426. DOI:http://dx.doi.org/10.1109/TR.2002.804489

Paul Menage. 2004. *CGROUPS*. Technical Report. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

Mesosphere. 2017. Marathon Health Checks. (2017). https://mesosphere.github.io/marathon/docs/health-checks.html

Mesosphere. 2018. Marathon API. (2018). https://docs.mesosphere.com/1.11/deploying-services/marathon-api/

G Meszaros. 1998. A pattern language for pattern writing. *Pattern languages of program design* (1998). http://xunitpatterns.com/~gerard/plopd3-pattern-writing-patterns-paper.pdfpapers2://publication/uuid/420266E9-5BD8-41A3-AA9B-F03763E9E78E

Microsoft. 2017a. Health Endpoint Monitoring pattern. (2017). https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring

Microsoft. 2017b. Microsoft Azure Scheduler. (2017). https://azure.microsoft.com/en-us/services/scheduler/

Adrian Mouat. 2015. *Docker Security*. Technical Report. http://www.oreilly.com/webops-perf/free/docker-security.csp

Netflix. 2011. The Netflix Simian Army. (2011).

Netflix. 2017. Chaos Monkey. (2017). https://github.com/Netflix/chaosmonkey

Andrew Odewahn. 2014. *Field Guide To The Distributed Development Stack*. Vol. 53. 160 pages. DOI:http://dx.doi.org/10.1017/CBO9781107415324.004

Pingdom. 2017. Pingdom. (2017). https://www.pingdom.com/

Eduardo Pinheiro, WD Weber, and LA Barroso. 2007. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, Vol. 7. 17–29. DOI:http://dx.doi.org/10.1016/j.engfailanal.2005.10.010

Pivotal. 2007. RabbitMQ Tutorials. (2007).

J Paul Reed. 2014. *DevOps in Practice*. 34 pages. DOI:http://dx.doi.org/10.1017/CBO9781107415324.004

Nathan Regola and Jean-Christophe Ducom. 2010. Recommendations for Virtualization Technologies in High Performance Computing. *Cloud Computing Technology and Science CloudCom 2010 IEEE Second International Conference on* (2010), 409–416. DOI:http://dx.doi.org/10.1109/CLOUD.2011.29

New Relic. 2017. New Relic. (2017). https://newrelic.com/

Chris Riley. 2015. Version Your Infrastructure. (2015). https://devops.com/version-your-infrastructure/http://devops.com/2015/11/12/version-your-infrastructure/

C Roderick, L Burdzanowski, and G Kruk. 2013. *The CERN Accelerator Logging Service- 10 Years in Operation: A Look at the Past, Present and Future*. Technical Report. CERN. http://cds.cern.ch/record/1611082

MJ Scheepers. 2014. Virtualization and Containerization of Application Infrastructure: A Comparison. (2014). http://referaat.cs.utwente.nl/conference/21/paper/7449/virtualization-and-containerization-of-application-infrastructure-a-comparison.pdf

Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. 2006. *Security Patterns: Integrating Security and Systems Engineering*. 600 pages.

Niels Seidel. 2017. Empirical Evaluation Methods for Pattern Languages: Sketches, Classification, and Network Analysis. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. 24.

Jens Smeds, Kristian Nybom, and Ivan Porres. 2015. DevOps: A definition and Perceived Adoption Impediments. *Lecture Notes in Business Information Processing* 212 (2015), 166–177. DOI:http://dx.doi.org/10.1007/978-3-319-18612-2{_}14

Stephen Soltesz, Stephen Soltesz, Herbert Pötzl, Herbert Pötzl, Marc E Fiuczynski, Marc E Fiuczynski, Andy Bavier, Andy Bavier, Larry Peterson, and Larry Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.* 41 (2007), 275–287. DOI:http://dx.doi.org/10.1145/1272998.1273025

Ivan Špeh and Ivan Heđ. 2016. A Web - Based IoT Solution for Monitoring Data Using MQTT Protocol. In *Smart Systems and Technologies (SST), International Conference on*. IEEE Computer Society, 249–253. DOI:http://dx.doi.org/10.1109/SST.2016.7765668

Statuscake. 2017. StatusCake. (2017). https://www.statuscake.com/

Darryl Taft. 2015. How the Skills Gap Is Threatening the Growth of App Economy. (2015). http://www.eweek.com/developer/slideshows/how-the-skills-gap-is-threatening-the-growth-of-app-economy.html

Saugatuck Technology. 2014. *Why DevOps Matters : Practical Insights on Managing Complex & Continuous Change*. Technical Report.

Tim Wellhausen and Andreas Fiesser. 2011. How to write a pattern? *Proceedings of the 16th European Conference on Pattern Languages of Programs - EuroPLoP '11* (2011), 1–9. DOI:http://dx.doi.org/10.1145/2396716.2396721

E O Winstedt. 1899. A Bodleian MS. of Juvenal. *The Classical Review* 13, 4 (1899), 201–205. http://www.jstor.org/stable/694154

Viveca Woods and Rob Meulen. 2016. Gartner Says Worldwide Public Cloud Services Market Is Forecast to Reach $204 Billion in 2016. (2016). http://www.gartner.com/newsroom/id/3188817

M G Xavier, M V Neves, F D Rossi, T C Ferreto, T Lange, and C a F De Rose. 2013. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* LXC (2013), 233–240. DOI:http://dx.doi.org/Doi10.1109/Pdp.2013.41

XebiaLabs. Periodic Table of DevOps. (????). https://xebialabs.com/periodic-table-of-devops-tools/

Graham Yarbrough and Sandy Hook. 2002. Message Queue Server System. (2002). https://www.google.com/patents/US20020004835

Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications* 1, 1 (4 2010), 7–18. DOI:http://dx.doi.org/10.1007/s13174-010-0007-6

A. PATTERN LANGUAGE FOR ENGINEERING SOFTWARE FOR THE CLOUD

A.1 Containerization

Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself. CONTAINERIZATION proposes the usage of containers to package the service and its dependencies and enable its isolated and programmatic deployment.

A.1.1 *Context.* Today's hardware, with multi-core and multi-CPU architectures, is built to execute multiple programs concurrently. Cloud computing often exploits resource sharing for executing multiple services in a single host. Sharing the host's operating system with the hosted services might introduce software incompatibilities between them or quickly clutter the host, as it must mutate its file system to accommodate each service's dependencies. Such introduced the need for isolated environments. Full stack virtualization quickly became the de facto standard approach to enabling resource sharing, allowing services to be executed in a dedicated installation of the operating system. Paravirtualization further improved that approach by exposing hardware resources directly to the virtualized environment. Still, isolation is achieved with an increased cost of hardware usage required to virtualize the operating system stack on each hosted environment.

A.1.2 *Example.* Consider a web application that has three services: an HTTP server, a database and an object caching service. These services share some core libraries, but each depend on different versions. The development team uses a few different Linux distributions for development but production environments are to use a specific distribution. All three services should be deployed on a temporary host for testing purposes and afterwards deployed in the production environment. It becomes a complex task to develop and deploy each service such that it is easily executed by each team member, as well as quickly installed in the development and production, despite existing configurations or the adopted distribution.

A.1.3 *Problem. Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself.*

Software deployments tend to couple services with their host environment, modifying it according to their needs [Koutoupis 2018]. When hosting multiple services that share resources, namely file-system, CPU, memory and network availability, unexpected behavior might be observed as they compete for those resources. Furthermore, situations exist where two services cannot coexist in the same environment due to incompatible dependencies, either virtual or physical.

A.1.4 *Forces.* The following forces, represented in Figure 2, need to be balanced while considering the adoption of this pattern:

Resource Management: Not using all the resources is a server is not cost-efficient, while over-allocating services will degrade their performance.

Overhead: Decoupling services from the operating system might lead to computation overheads.

Supervision: The service status must be monitored, triggering a recovery on failures.

Isolation: Installation of dependencies changes the host, possibly resulting in side effects with other services in the same host.

Portability: Programmatic system deployment requires the packaged software to be easily deployed in different environments.

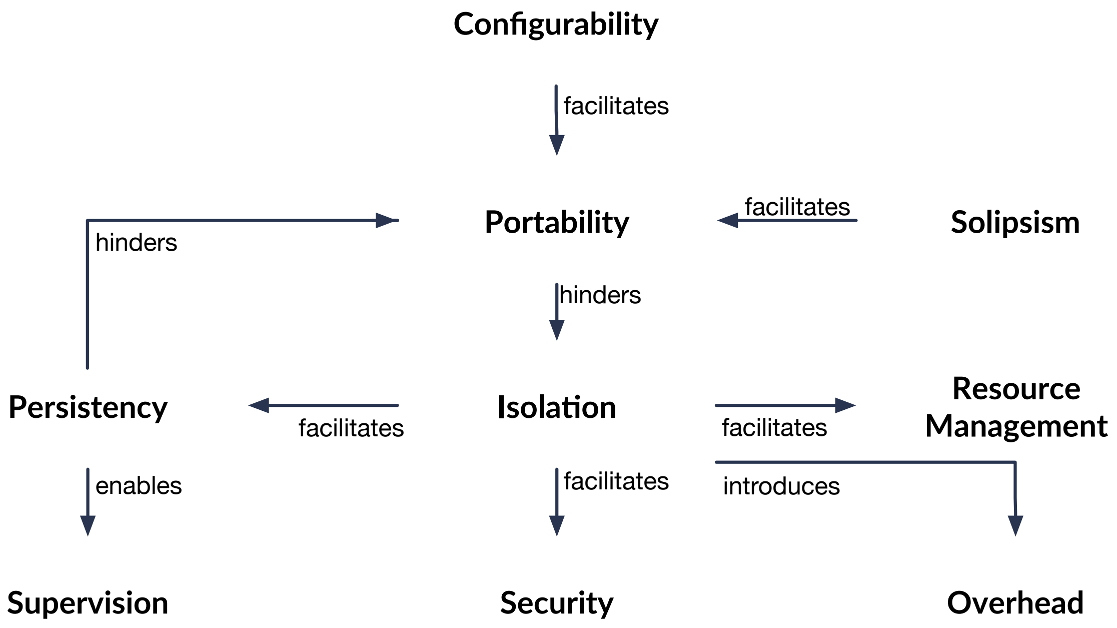


Fig. 2: Relationship between CONTAINERIZATION forces.

Configurability: Programmatic system deployment requires a strategy for configuration in execution time.

Security: Different approaches to isolation introduce different levels of security by default.

Solipsism: Each running environment should only manage itself, communicating with external services resiliently.

Persistency: Persist data in the host beyond the service's execution lifetime, possibly being reused in future executions.

A.1.5 *Solution. Use a container to package the service and its dependencies and enable its isolated programmatic deployment.*

Full stack virtualization provides isolated environment for running software. Despite that, the cost of virtualizing the operating system for each environment introduces considerable overheads in CPU, memory. Portability is also limited, given the increased disk usage. As such, this approach is not optimal solution for cloud software.

A better solution exists in operating system level virtualization, also known as containers. A container is a self-contained isolated environment with a virtual file-system, network and resources allocation which is executed within an host operating system [Soltesz et al. 2007].

The container can be created and started programmatically, with configurations provided to the inner software as environment variables, making it portable between hosts. Strict resource allocation ensures that the container will not overuse the available hardware resources. Figure 3 demonstrates how to configure and print environment variables for a container.

Persistent storage can be setup in the container by exposing files or folders from the hosting server inside the container. File system access is limited to those. When the container is deleted from the host, all its data is deleted as well, leaving behind only the files and folders created in the in the exposed storage to the container, if any.

On failure, it can restart itself with the same configurations and a clean environment.

There are multiple container implementations available today, with Docker[1] being the most adopted.

A.1.6 *Example Resolved.* Each service would be packaged into a separate container. In a development environment, the three containers could be started in the same host. A separate production environment could have each container being executed in an independent host. No changes would have to be made to the containers, other than starting them with the proper configuration as environment variables, which can easily be automated.

If needed, each service can be scaled independently from the others by increasing the number of instances for that specific container.

A.1.7 *Resulting Context.* This pattern introduces the following benefits:

—Resource use is optimized, with overheads being decreased when compared to full stack virtualization, as only a thin layer needs to be virtualized, improving the performance achievable by a host.

—Resources can be allocated to the container, leveraging the available host's resources between multiple containers, as well as what is exposed from the container to the host and vice-versa.

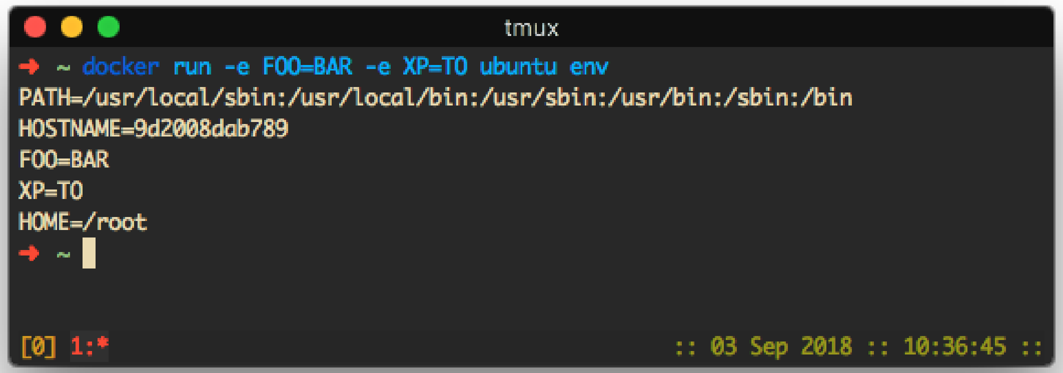
—Arguments can be provided to the container on execution to configure the service running inside it. Due to its immutability, in case of failure the container can restart with the original configuration.

—Isolated environment can be easily ported between development and production as the image size only packages the service and its dependencies, leaving out all operating system's components.

The pattern also introduces the following liabilities:

—Paravirtualization is a virtualization technique that exposes part of the host's hardware directly to the virtual machine. In some low-level hardware access scenarios, paravirtualization might provide increased performance.

[1]Learn more about docker at `https://www.docker.com/`.



```
➜ ~ docker run -e FOO=BAR -e XP=TO ubuntu env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=9d2008dab789
FOO=BAR
XP=TO
HOME=/root
➜ ~

[0] 1:*                                    :: 03 Sep 2018 :: 10:36:45 ::
```

Fig. 3: Running a containerized Ubuntu image with injected environment variables. Environment variables are provided using the *-e* argument. This example executes the ENV command and exits, which simply prints the environment variables. Environment variables can be read by software running inside the container as a way of providing runtime configurations.

—Packaging services as containers will still introduce overheads when compared to installing services directly in the host.

A.1.8 *Related Patterns.* Configuration might be required for a container to be adaptable to multiple hosts and scenarios. Using the Environment-based Configuration pattern it is possible to use environment variables to configure running services at execution time.

Some containers might have the need to persist information between executions in the host. That is the case of isolated databases that cannot lose their data if the machine reboots. With this goal in mind, the Local Volumes pattern may be used to expose a folder from the host inside the container.

A.1.9 *Known Uses.* Containerization was first introduced in 1982 in the Seventh Edition Unix by Bell Labs, as a tool for testing the installation and build system of the operating system, providing an isolated file-system environment where services could be executed. By 2008 Linux Containers (LXC) were introduced in Linux Kernel version 2.6.24, reducing the virtualization overhead and increasing efficiency [Felter et al. 2014]. By 2013 Docker was built, based on LXC, in order to make containerization easier for a broader audience.

Docker is now the cloud standard for container-based deployment, with native support with multiple cloud providers, such as Amazon Web Services and Google Cloud Platform, both with native support for running docker containers [Amazon 2015; Google 2015]. A draft is being worked on to create a standard format for containers, with RunC being the reference implementation for it, which can also run Docker-created containers [Initiative 2015].

A study by DataDog in April 2018 showed that almost 25% of their clients were already using containers, with about 50% using some sort of Orchestration Manager [DataDog 2018].

A.1.10 *Discussion.* While container adoption is rising, virtual machines will always be part of cloud computing as the unit of provision of computation. For the development team, the question at hand is if services should be deployed at the virtual machine or container level, what their differences are and how to decide. This section sheds some light over this decision. Given the specific context of cloud computing, deploying natively is not within the scope of this discussion.

Providing some context over virtualization, it is built by leveraging a hypervisor to create and execute virtual machines. Hypervisors are responsible for the virtualization of the hardware in a virtual machine and are available in two different flavors: those who run on bare metal, such as Xen, and those who require an underlying operating system such as KVM[2]. In both scenarios, a virtual machine is a fully virtualized computing environment, meaning that every hardware component the virtual machine would see, namely the CPU, RAM or graphical card, would in fact be a virtual representation of such element. It is part of the hypervisor responsibility to them map those virtual components to the actual ones available.

Containers work differently, by having the hosted services sharing resources with the host environment, with the actual service execution being managed by the host's kernel, although in an isolated environment.

A.1.10.1 *Performance.* Performance is key in any system. Virtualization efficiency is typically inverse to the overhead introduced by the virtualization system. As previously described, each virtual machine requires its hypervisor to virtualize the hardware and operating system layers, which introduces an immense overhead. As such, virtualization is less efficient than containers. In fact, containers provide almost no overhead when compared to running in bare metal given that they actually share their host's operating System kernel and, at time, binaries and libraries as well. Theoretically, containers are a much more efficient solution to deploy multiple isolated environments in a server.

[2]Xen and KVM are both open source virtualization servers. Learn more about the projects at `https://www.xenproject.org/` and `https://www.linux-kvm.org/`.

This theory has been validated by Xavier, whom made an extensive evaluation of native systems performance when compared to three container implementations (LXC, OpenVZ and VServer[3]) and the aforementioned Xen virtual environments [Xavier et al. 2013], visually represented in figure 4.

Regarding computing performance, Xavier concluded that there were no statistically significant differences between native and the container implementations, but observed a 4.3% overhead with Xen virtualization.

The same study evaluated the memory performance of these three systems and also concluded that containers have similar performance to native, but observed a 31% overhead with Xen based virtualization. The author identifies this overhead to be a product of the hypervisor layer responsible for virtual machine to native memory address translation.

Finally, regarding disk IO, again containers presented a similar performance to native, with OpenVZ actually outperforming native. Xen on the other hand presented poor results with read and write performance being about 50% when compared to native.

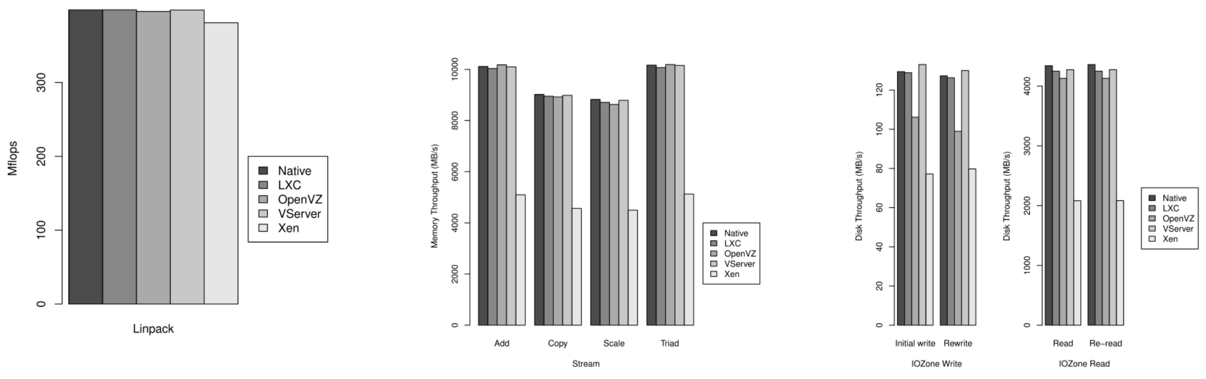
A.1.10.2 *Resource Isolation.* When running multiple virtualized or containerized services in a server, they shouldn't negatively impact the performance of their neighbors. Such is possible by setting hard-limits on resource usage.

With Xen, resource allocation is a requirement for the creation of the virtual machine. These resources are reserved by the hypervisor, which will only expose to the virtual machine the allocated resources.

Containers typically rely on the Linux Kernel Control Groups (cgroups) to enforce resource allocation. Control Groups allow the creation of a resource pool to be allocated to a given subsystem, enabling resource attribution to those. In practice, it limits the resources available to a service and it's descending processes [Menage 2004].

Enforcing resource limitation introduces an overhead per se, which might have impact remaining existing systems. In his research, Xavier ran more than one virtualized or container systems, with one trying to use more resources than the ones allocated. He observed that for both Xen and LXC, CPU limitation is effective, not imposing any performance impact on the other hosted system. The same is not true for memory management, with the Xen hosted service having a minimal 0.9% performance impact, but with LXC presenting an impact of

[3]LXC, OpenVZ and Vserver are three alternative container implementation. LXC was used internally by Docker until version 0.9, being replaced by lib-container since. You can learn more about these projects respectively at `https://linuxcontainers.org/`, `https://openvz.org/` and `http://linux-vserver.org/`



(a) Computing performance using Linpack for matrices of order 3000.

(b) Memory throughput using STREAM.

(c) Disk throughput using IOZone.

Fig. 4: Comparison of (a) computation performance, (b) memory management and (c) disk throughput, from Xavier's work.

88.2% [Xavier et al. 2013]. Several other studies showed similar results, demonstrating that containers introduce negligible performance impact [Soltesz et al. 2007; Regola and Ducom 2010; Felter et al. 2014].

A.1.10.3 *Security.* Security is essential when executing services inside isolated environments. The service should not be able to access its host unless explicitly configured to do so.

Virtual machines, by design, provide optimal security to the host. A service running inside a virtual machine will not be able to understand if it is executing in a native or virtualized environment.

Opposed to virtual machines, containers do present an increased security thread. Given that the containerization engine is executed by the host's operating system kernel and that it requires *root* permissions, the kernel itself becomes an attack vector. In the Docker Security report [Mouat 2015], the author listed a set of security measures recommended for container administrators, namely ensuring that the host's kernel is always using the latest version and that hosted containers are from trusted sources and do not present security flaws in them and that programs within them always executing using the least privilege possible, meaning that they should only have the required permissions to execute their functions.

A.1.10.4 *Flexibility.* Virtual machines provide the most flexibility for hosts and hosted environments. Given the existence of an hypervisor for a given machine, it will be able to create virtual machines and host any operating system with compatible architecture within it. As for containers, they currently only run natively in Linux systems, requiring some sort of virtualization in other operating systems to execute. Furthermore, and focusing on the Docker implementation, containers will only Linux as well [Bui 2015].

A.1.10.5 *Conclusion.* We can conclude that container environments are still more prone to security flaws than virtual machines. New techniques for securing containers have been made available recently and more are expected to become available in the future, but it is imperative that the user acknowledges the problem and evaluate its risks while using containers.

A.2 Orchestration Manager

Deploying and updating software at scale is an error-prone, slow and costly process. Such can be facilitated by adopting an ORCHESTRATION MANAGER to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements.

A.2.1 *Context.* Along with cloud computing came very large applications, typically composed by several services, that needed to scale out to multiple servers.

Traditional teams would have an operations team that would deploy and operate the software built by the development team. This approach revealed itself impractical due to slow deployments and recurrent conflicts between the two teams [De Bayser et al. 2015] due to miscommunication and finger pointing. DevOps suggested merging both teams, having a single team responsible for the software life cycle. For that to happen, operations needed to be fully programmatic [De Bayser et al. 2015].

For achieving this level of automation, abstractions where required to facilitate building fully automated operation strategies. CONTAINERIZATION played an essential role in enabling programmatic deployment of software.

A.2.2 *Example.* An application is composed by two services that need to be orchestrated in an infrastructure with four servers. The service requirements might change with time and must be allocated into suitable hardware. Their current requirements are described in Table I.

The servers might also change with time, with more powerful or specialized hardware being allocated if need. The current servers available are described in Table II.

Service Name	CPUs	RAM	Disk Space	Instances	Constraints
HTTP	2	2 GB	5 GB	4	hostname=unique; location=Europe
Database	2	8 GB	50 GB	2	hostname=unique; SSD=true; location=Europe

Table I. : List of services and their possible configurations for a production environment.

Server name	CPUs	RAM	Disk Space	Server Details
Alpha	4	4 GB	500 GB	location=Europe
Beta	4	4 GB	500 GB	location=Europe
Charlie	4	16 GB	1000 GB	SSD=true; location=Europe
Delta	4	16 GB	1000 GB	SSD=true; location=Europe

Table II. : List of servers available in the infrastructure, along with their meta-data.

A.2.3 *Problem. Deploying and updating software at scale is an error-prone, slow and costly process.*

Multiple variants can constraint the allocation of services to servers in an infrastructure. Each service has its own requirements and each service provides a specific set of resources. Furthermore, given the wide adoption of continuous integration and deployment strategies, teams are increasing the frequency at which they deploy their services to several times per day [Cycligent], which demands automation in the deployment process.

A common requirement is to ensure that services are allocated to host machines which fulfills its hardware requirements and that this happens without human interaction. Such enables servers to run multiple services while ensuring their execution within the host's resource limits, guaranteeing the expected performance.

Cloud applications can also scale and the infrastructure empowering it must facilitate such scaling as well to adapt to a change in the volume of activity, while optimizing costs.

A.2.4 *Forces.* The following forces, represented in Figure 5, need to be balanced while considering the adoption of this pattern:

Infrastructure Decoupling: The development process should not be constraint by the running environment.

Resource Allocation: Allocating services without ensuring their requirements will result in unexpected behavior.

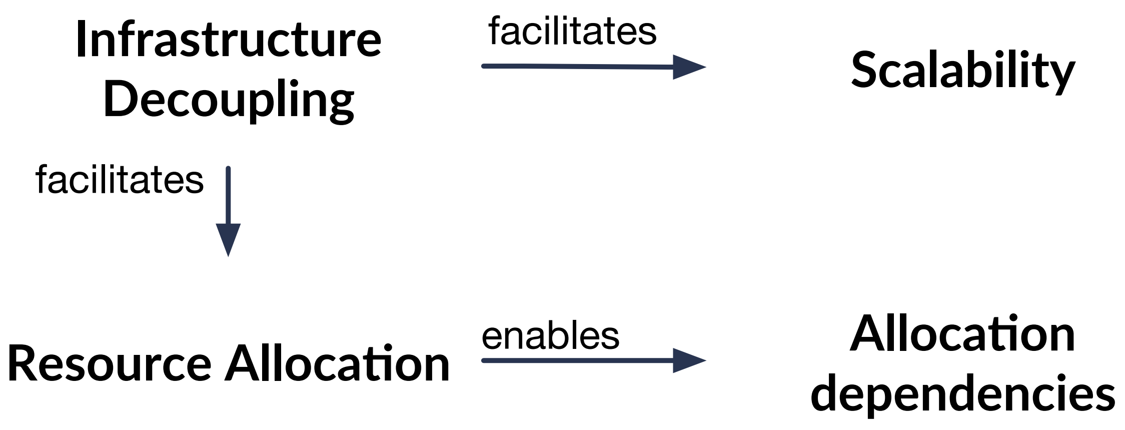


Fig. 5: Relationship between ORCHESTRATION MANAGER forces.

Allocation dependencies: Allocating services without ensuring their dependencies will result in unexpected behavior.

Scalability: It must be possible to scale the system either up or down.

A.2.5 *Solution. Adopt an* ORCHESTRATION MANAGER *to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements.*

Adopting an ORCHESTRATION MANAGER provides abstraction and automation over the orchestration of services. The abstraction is provided by having the ORCHESTRATION MANAGER evaluating each available server, service and its requirements and use that information to optimize service allocation. Automation is provided by exposing a programmatic interface that facilitates orchestrating software in the infrastructure.

Services can be deployed programatically after being packaged using CONTAINERIZATION. Using a declarative strategy, the ORCHESTRATION MANAGER can be told what services need to be deployed and their requirements, leaving to it the responsibility of managing the allocation. Resource allocation is enforced, ensuring that all services are provided their required resources to execute properly. Most ORCHESTRATION MANAGERS enable the specification of additional restrictions such as co-allocations or startup sequences. Listing 1 demonstrates how to tell the Kubernetes ORCHESTRATION MANAGER to instantiate two Nginx web servers [Kubernetes].

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Listing 1: A Kubernetes specification for starting two instances of the Nginx web server.

ORCHESTRATION MANAGERS work using a master-slave architecture, being the master elected automatically and responsible for handling service allocation. Deployment requests can often be issued to any slave, which proxies them to the master [Mesosphere 2018]. This approach facilitates electing a new master automatically if the current master fails.

Whenever a new slave joins the infrastructure, the master identifies its available resources. When a new service allocation request is received, the master decides where the service should be executed and instructs the slaves to start it. Figure 6 illustrates this interaction.

If no slave is capable of hosting the service due to a mismatch on the service requirements and those available in the servers of requirements, the master periodically retries the service allocation until it succeeds.

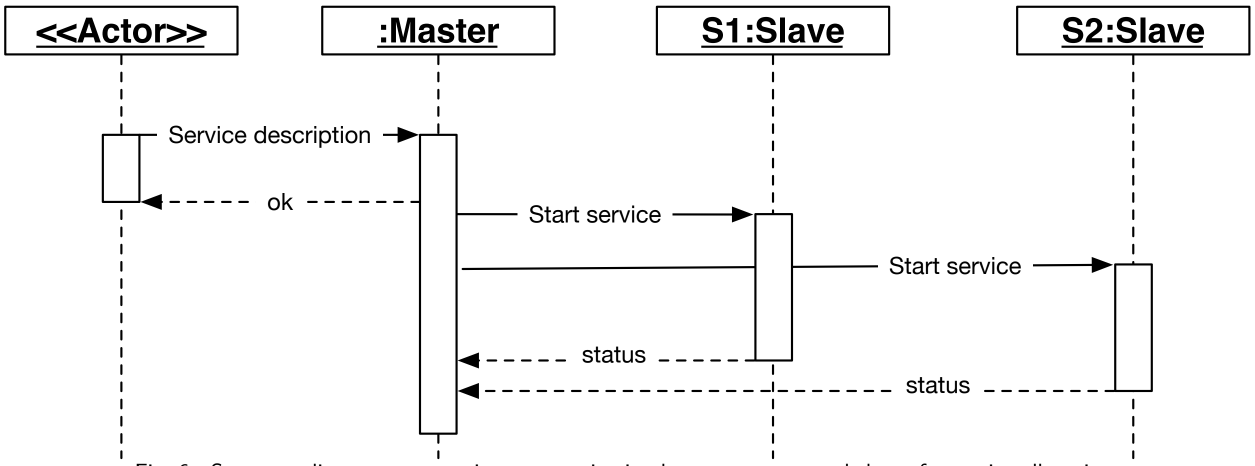


Fig. 6: Sequence diagram representing communication between master and slaves for service allocation.

A.2.6 *Example Resolved.* The team starts by deploying an Orchestration Manager that abstracts four existing servers. By doing so, one of the servers will be automatically elected as master, with the others proxying orchestration requests to it.

A descriptive file can be created for each service, describing how to obtain the respective container, as well as describing its requirements.

Finally, to deploy the services, a request similar to the one from Listing 1 is sent to the Orchestration Manager with each service description. The Orchestration Manager master evaluates the resources required by the services and the ones available in each server, instructing the selected servers to deploy the services.

In the example we can see that the hostname must be unique, meaning that it is not possible to deploy two HTTP or database servers in the same host. Also, the selected servers must be Europe, with the database service in a server with SSD storage.

Considering those restrictions, the Orchestration Manager would compute a viable solutions, which could be the one identified in Table III.

In this example, when deploying the services, all servers are at full capacity and are able to fulfill the requested resources regarding CPU, RAM and disk space. When the service is deployed to Alpha and Beta, the orchestration Manager subtract 2 CPUs, 2 GB RAM and 5 GB of storage from their available resources, influencing the allocation of services in the future. When deploying the database service, only European servers with SSD storage can be used, resulting in Charlie and Delta being the two only eligible hosts.

A.2.7 *Resulting Context.* This pattern introduces the following benefits:

Infrastructure decoupling: Service development can be agnostic of the host where the service is going to be placed, describing only its requirements and packaging its dependencies using containerization.

Service Name	Server Name	Applied constraints
http server	Alpha	hostname=unique; location=Europe
http server	Beta	hostname=unique; location=Europe
database	Charlie	hostname=unique; SSD=true; location=Europe
database	Delta	hostname=unique; SSD=true; location=Europe

Table III. : List of available servers in the infrastructure.

Resource allocation: Services are allocated in servers that meet their requirements.

Allocation dependencies: Dependencies are respected, managed as constraints for the allocation process.

Scalability: Scalability is achieved by adding slaves to the infrastructure and individually change the number of instances for each service.

The pattern also introduces the following liabilities:

Suboptimal allocation: Allocation using a greedy placement algorithm might result only in a locally-optimal solution.

Single point of failure: In some implementation where the master is not automatically reelected in case of failure, using a single master node would result in a single point of failure.

A.2.8 *Related Patterns.* Some ORCHESTRATION MANAGER implementations might support additional strategies for running software, but CONTAINERIZATION is the most common strategy.

A.2.9 *Known Uses.* Kubernetes by Google is the fastest growing implementation of a ORCHESTRATION MANAGER. It abstracts a set of machines, receiving requests for allocating containers in the infrastructure. Kubernetes is under active development, widely adopted and supported across most cloud providers [Goasguen 2016].

Mesos and Marathon together provide another robust solution for achieving the same goal. New services are submitted to the infrastructure using an HTTP API describing its requirements and constraints. With this information, the master communicates with the slaves, identifying a valid host and issuing the order for placing the service [Hindman et al. 2011].

CoreOS offers similar technology, with a centralized registry made available using Etcd [CoreOS Community 2015].

A.3 Automated Recovery

Services may fail during execution and need to be recovered in a timely and orderly fashion. Including health checks and recovery configurations in the instructions used for the ORCHESTRATION MANAGER to orchestrate containers, enables it monitor and recover failing containers.

A.3.1 *Context.* At the scale that cloud software is operated, it is reasonable to accept that it will eventually fail. Resilience is then an essential requirement while writing scalable cloud software. The development team must introduce the necessary strategies to ensure that the application is functioning properly or that, at least, it can recover back to a functioning state automatically.

This pattern extends the ORCHESTRATION MANAGER [Boldt Sousa et al. 2015] pattern, responsible for executing services packaged using CONTAINERIZATION [Boldt Sousa et al. 2015].

A.3.2 *Example.* Consider a web server exposing an API is running inside a container in an ORCHESTRATION MANAGER. Suppose that the service had a memory leak, which gradually consumed the memory allocated for the service, and thus making the service unresponsive. The ORCHESTRATION MANAGER sees the container running, but while it is still executing, it is unable to respond to requests.

A.3.3 *Problem. Services may fail during execution and need to be recovered in a timely and orderly fashion.*

Cloud software is exposed to variety of stress conditions, from public Internet exposure to dynamic cloud infrastructure. As such, software should be designed with resilience in mind to ensure it can to recover from failures.

With a traditional operations approach, a team member is responsible for identifying failures and deciding the best action to recover a failing system using the defined recovery protocol. This approach is troublesome as it requires manual intervention, which is slow and error prone.

A.3.4 *Forces.* The following forces, represented in Figure 7, need to be balanced while considering the adoption of this pattern:

Resilience: Failing containers should recover to an healthy state when failure is observed.

Reliability: Monitoring strategies that are prone to false positives can trigger an unnecessary service recovery.

Automation: Requiring manual intervention for recovering a failing service is error-prone, slow and costly.

A.3.5 *Solution. Including health checks and recovery configurations in the instructions used for the* ORCHESTRATION MANAGER *to orchestrate containers, enabling it monitor and recover failing containers.*

AUTOMATED RECOVERY is available in most ORCHESTRATION MANAGER implementations [Mesosphere 2017; Kubernetes 2018]. The development team implements health checks for each container to verify if its service is behaving correctly. Most implementations provide at least plain TCP and HTTP checks. The health checks can be provided along with the service description directly to the ORCHESTRATION MANAGER.

To implement the recovery strategies the team needs to evaluate each service individually, deciding which check can be used to identify that the service is failing, how many times each check needs to be retried and how much time to wait between executing the checks and actually considering a service as failing. A recovery protocol must be made available along with the health checks to be automatically executed by the ORCHESTRATION MANAGER to attempt the service recovery. Health checks and recovery protocols need to be considered part of the service's development process.

Health checks will be very specific to the service running in a container. These might range from checking if a port is receiving connections in the container, to more a advanced HTTP-based checks, to executing a command inside the container and monitoring its exit code.

TCP checks verify if a network port is open and accepting TCP connections. These are typically binary checks that just validate the service ability to receive connections.

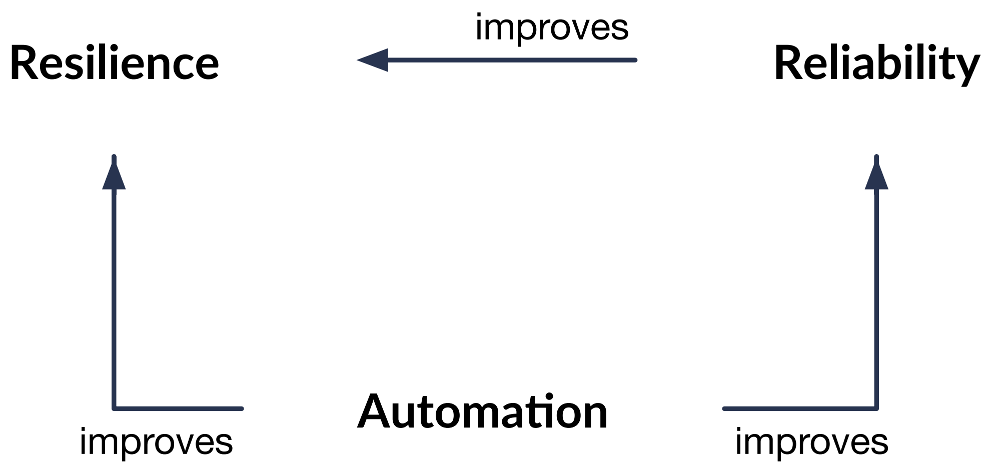


Fig. 7: Relationship between AUTOMATED RECOVERY forces.

HTTP checks are more advanced than their TCP counterparts since they can actually make HTTP requests and validate the HTTP return code and body, making way for more advanced tests.

Developers can implement dedicated health checking endpoints to be queried by AUTOMATED RECOVERY, providing responses that can be easily interpreted to verify the service's status.

While deciding on the supervision strategy, the team can leverage common features that prevent false positives from mistakenly restarting the service. A false positive might be a momentary request that fails to have a response, followed by normal service operation.

The recovery operation itself is prone to failure. Implementing this pattern is another step towards improving cloud software reliability, but cannot be relied upon as unbreakable.

When a failure is identified and that results in an AUTOMATED RECOVERY, the ORCHESTRATION MANAGER or the adopted AUTOMATED RECOVERY service will log that event and its details. The team can use this log as input for improving their software or to configure notifications to be aware as soon as they happen. This is relevant as restarting the container might only be a temporary solution or not able to fix the problem.

```json
{
  "id": "toggle",
  "container": {
    "docker": {
      "image": "busybox"
    }
  },
  "cpus": 2,
  "mem": 32.0,
  "healthChecks": [
    {
      "protocol": "HTTP",
      "path": "/health",
      "portIndex": 0,
      "gracePeriodSeconds": 5,
      "intervalSeconds": 10,
      "timeoutSeconds": 10,
      "maxConsecutiveFailures": 3
    }
  ]
}
```

Listing 2: A Marathon service description, describing the health check policies for AUTOMATED RECOVERY.

Listing 2 demonstrates how a service can be started using the Marathon ORCHESTRATION MANAGER, configuring an HTTP health check that verifies the response code from the */health* endpoint. From this example, the parameters used are: *gracePeriodSeconds*, which ignores errors for a given number of seconds after the service starts; *intervalSeconds*, which configures the delay between checking the endpoint; *timeoutSeconds*, which configures the maximum time to wait for a response from the service; and *maxConsecutiveFailures*, which defines the number of times the health check can fail before being restarted.

While implementing this pattern, one needs to decide on how to balance:

Interface coverage: We want to ensure the tests are as complete as possible, covering all application's interfaces, while balancing this investment with the available development effort.

Frequency: We want to run the health checks as often as possible, while balancing this frequency with the increase in resource utilization.

Accuracy: We want to prevent false positives by confirming issues redundantly, such as those who might result from a temporary slowdown in the system, automatically recovered inside the container.

A.3.6 *Example Resolved.* While deploying a service with an ORCHESTRATION MANAGER with support for AUTOMATED RECOVERY, the service definition specifies the set of health checks used to verify the service's status.

During execution, if a health check identifies a problem with a container, the respective container is restarted automatically. While the intrinsic issue might persist, the service will once again become available without team intervention. Given the notification sent to the team, they will be immediately aware of the issue and can focus on implementing a proper solution.

A.3.7 *Resulting Context.* This pattern introduces the following benefits:

Resilience: The ORCHESTRATION MANAGER will be able to recover failing containers automatically.

Reliability: Failing containers will be automatically identified using the implemented health checks, which can have an advanced strategy to prevent false positives.

Autonomy: Failing services are restarted using the implemented recovery protocol, so that the system recovers its correct execution state automatically and without requiring manual intervention.

On the other hand, the following liabilities are also introduced:

Relaxation: It might happen that the development team disregards software failures given that they are being automatically recovered.

Unawareness: Without the proper monitoring and logging in place, considering that failing services are automatically recovered, it might happen that the team isn't aware of the failure in the system.

Performance degradation: Running health checks against the container will introduce additional load in the system, which might result in performance degradation.

False positives: It might happen that the health checks aren't accurate and the containers restarted while behaving correctly.

Unintended Consequences: It might happen that the service is improperly designed and unable to be restarted, leaving it inconsistent and requiring manual intervention after a restart. In extreme scenarios each recovery attempt might further increase the problem. An example of such is the case where a backup system that is consistently failing during its execution will keep increasing the disk space it occupies without ever actually having a complete backup, until no more space is available.

A.3.8 *Related Patterns.* The ORCHESTRATION MANAGER pattern describes how containers can be orchestrated in an infrastructure automatically, leveraging allocation rules, container scaling and resource availability. AUTOMATED RECOVERY is commonly related with ORCHESTRATION MANAGER, given that most of its implementations provide some sort of supervision strategy to ensure the containers are working as expected [Mesosphere 2017; Kubernetes 2018].

AUTOMATED RECOVERY enables the automatic recovery of services if their health is degraded. This pattern is essential to implementing FAILURE INJECTION, where the reliability and resilience of the system is tested though a set of random inputs and events in order to identify possible attack vectors or failures in the system.

Dynamic Failure Detection and Recovery describes a subset of AUTOMATED RECOVERY, by proposing the existence of a resilient watchdog component that monitors IT resources and in case of failure notifies the team and attempts automated recovery [Arcitura Education Inc].

A.3.9 *Known Uses.* Most Orchestration Manager pattern implementations provide Automated Recovery natively, as orchestration and supervision complement each other while deploying services in an infrastructure.

Marathon supports multiple health check strategies. TCP and HTTP are implemented as described in this pattern's solution. Additionally, Marathon supports the *COMMAND* check, which consists on running a command within a container and evaluate its output [Mesosphere 2017]. Listing 2 demonstrates how an health check can be configured for Marathon.

Kubernetes provides a similar approach to Automated Recovery [Kubernetes 2018], but with additional features to it. With Kubernetes, developers can set two flavors of health checks: readiness and liveliness. Readiness checks are considered only right after the container is instantiated and enable Kubernetes to check if the container is ready to start accepting traffic. Only after the readiness checks pass is the container considered healthy and ready to be used. Liveliness then work as health checks do in Marathon, periodically testing the container for its status, automatically restarting it when unhealthy.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Listing 3: A Kubernetes service description, describing the health check policies for Health Check using HTTP.

```
from nginx

HEALTHCHECK --interval=5m --timeout=3s CMD curl -f http://localhost/ || exit 1

CMD nginx -g "daemon off;"
```

Listing 4: A Dockerfile for building a container based on the Nginx image, leveraging Docker's implementation of Automated Recovery by periodically checking the web server's health.

Just like with Marathon, health checks are defined with the service definition, along with the specification of what container to use and how to configure it, as demonstrated in Listing 3.

Docker has its own supervision mechanism. Docker's supervision provides a simple restart strategy, which automatically restarts a container either if it fails or when an health check is failing. Health checks can be specified while creating the container [Docker 2018]. Docker health checks consist on the periodic of a command within the container, verifying its exit code. Listing 4 demonstrates how to create a Dockerfile which builds a Docker image using the Nginx image has base and simply executes Nginx on start, verifying every five minutes if the web server is responding to requests. If a request takes longer than three seconds to respond, the health check fails and the container is automatically restarted.

A.4 Job Scheduler

Cloud applications require frequent short-running jobs to be scheduled, which must be orchestrated across a dynamic infrastructure without permanently allocating resources. A scheduler service running along with the ORCHESTRATION MANAGER can instruct it to allocate one time or periodic jobs, recovering their resources to the infrastructure when they complete.

A.4.1 *Context.* It is often required that jobs are executed on a periodic basis inside an infrastructure managed by an ORCHESTRATION MANAGER. These jobs can range from internal system verifications, maintenance, infrastructure scaling and many others. These are not long running services, hence, do not need to be continuously executing on the infrastructure, as doing so preallocates valuable resources that would be idle part of the time.

In a non-cloud context, job scheduling was typically provided by Cron (see subsubsection A.4.10) or similar application. In the context of the cloud Cron is not a viable option, given that it is local to a specific server and not aware of the whole infrastructure and its resource availability.

This pattern considers the adoption of CONTAINERIZATION for packaging the jobs to execute and the presence of an ORCHESTRATION MANAGER.

A.4.2 *Example.* Consider a distributed database, replicated between multiple servers. Despite the replication, keeping frequent backups in a secure remote location is relevant to recover the database from an unexpected scenario in the infrastructure. This backup must happen frequently and automatically, without the team's intervention.

A.4.3 *Problem. Cloud applications require frequent short-running jobs to be scheduled, which must be orchestrated across a dynamic infrastructure without permanently allocating resources.*

It is common for short-running jobs to be executed in a infrastructure, alongside the hosted microservices. These can vary from database backups to internal system checks. Traditionally, these operations would be the responsibility of the operations team. Some degree of automation could be achieved by leveraging a job scheduler, such as Cron. In the cloud using Cron is not ideal given that the infrastructure is continuously evolving, that containers are dynamically allocated to their host servers and that co-location with specific containers or resource allocation rules might exist for running these jobs. Also, using Cron while using CONTAINERIZATION would require a container to be running for the sole purpose of executing scheduled jobs, permanently reserving resources for the container, or using the host's Cron scheduler polluting the host, both less than ideal approaches.

A.4.4 *Forces.* The following forces, represented in Figure 8, need to be balanced while considering the adoption of this pattern:

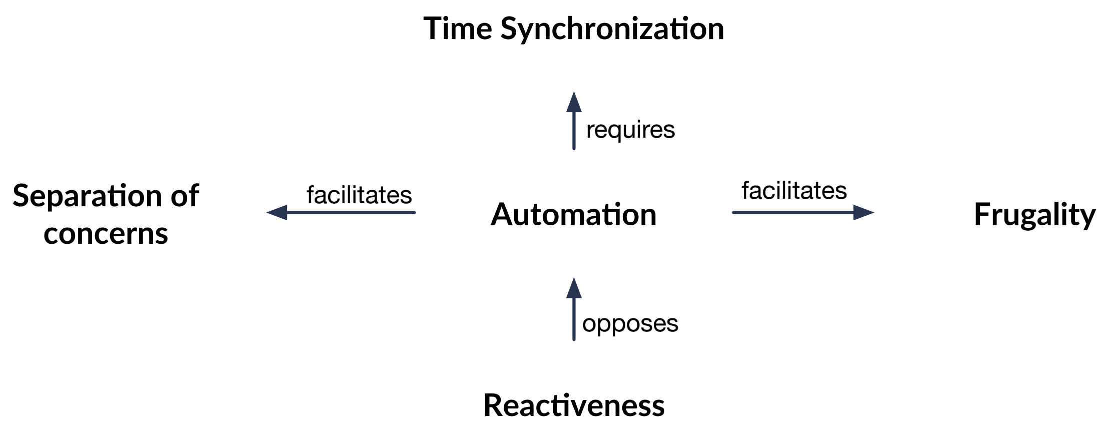


Fig. 8: Relationship between JOB SCHEDULER forces.

Automation: Manual intervention is error-prone, slow and costly.

Frugality: Permanent resource allocation to idle containers that are only active periodically is not resource efficient for the infrastructure.

Reactiveness: Some short-running jobs need to execute as a reaction to an external event (typically called triggers).

Separation of concerns: Short-running jobs are bundled with the description of the resources they require to execute, without needing to know anything about the infrastructure where they will be executed.

Time synchronization: Maintain machine clocks synchronized across the infrastructure to ensure that jobs are started at the correct time, despite what machine is starting the execution.

Execute one time or periodic jobs in the infrastructure.

A.4.5 *Solution. Deploy a scheduler service along with the* ORCHESTRATION MANAGER *that can instruct it to allocate one time or periodic jobs, recovering their resources to the infrastructure when they complete.*

A JOB SCHEDULER extends the ORCHESTRATION MANAGER pattern, responsible for executing services using CONTAINERIZATION, by enabling the scheduling and execution of one time or periodic jobs in the infrastructure. The pattern can be implemented by using a third-party JOB SCHEDULER that already integrates with the adopted ORCHESTRATION MANAGER.

The JOB SCHEDULER service can expose a programmatic, graphical or both, configuration interface to manage job scheduling. A job specification is composed by the instructions required to execute the job, along with its resource requirements and schedule details.

The exact information required for executing jobs will be specific to the adopted JOB SCHEDULER implementation, but will typically require the details of a container image to execute, along with list of environment variables to configure it, supervision criteria and required execution resources such as the required number of CPU cores or amount of RAM, just like any service would.

The JOB SCHEDULER should integrate with an ORCHESTRATION MANAGER, which is responsible for executing the scheduled job inside a container, honoring its requirement constraints. It does so by instructing the ORCHESTRATION MANAGER to execute a container for running the Job, while also providing the requirements for running it.

Allocated resources are freed upon the job completion, becoming available for executing other jobs. The integration with the ORCHESTRATION MANAGER ensures that jobs are only started if their required resources are available and restarted in case of unexpected failure, observed through the job's exit code.

An ORCHESTRATION MANAGER might also provide the possibility for restricting where jobs are executed in the infrastructure, by tagging the available servers and limiting allocation to servers who are tagged with a particular set of labels.

To ensure consistent behavior despite in which node the SCHEDULER is deployed, the hosts should have their clocks synchronized using an external time server.

A.4.6 *Example Resolved.* Deploy the scheduler service within the infrastructure. The backup operation would be configured in the scheduler to execute every day. The ORCHESTRATION MANAGER would be responsible for ensuring that the container responsible for executing the job is placed in a server that provides the required resources to run the job, as well as, if needed, it is co-located with the server running the database, reducing network latency.

A retry mechanism can also be specified, ensuring that the backup job would automatically retry up to a certain number of times in case of failure. If the failure persists, the execution of the job is aborted and the team is notified of the issue.

To ensure that all machines share the same date and time, a time synchronization daemon should be present.

A.4.7 *Resulting Context.* This pattern introduces the following benefits:

Automation: Jobs are automatically spawned on the infrastructure on their scheduled times, without requiring manual intervention.

Frugality: Resources allocation is minimized for short-running jobs, being recovered by the infrastructure once the job finishes.

Separation of Concerns: The scheduled job does not need to know details about the infrastructure, only describe its requirements. The ORCHESTRATION MANAGER will assume the responsibility of placing the container in the right host.

On the other hand, the following liabilities are introduced:

Dependency: If a job depends on another service's status, if that service is degraded, the job may consistently fail.

Single point of failure: When the scheduler fails, the ORCHESTRATION MANAGER will not be instructed about the jobs it needs to execute.

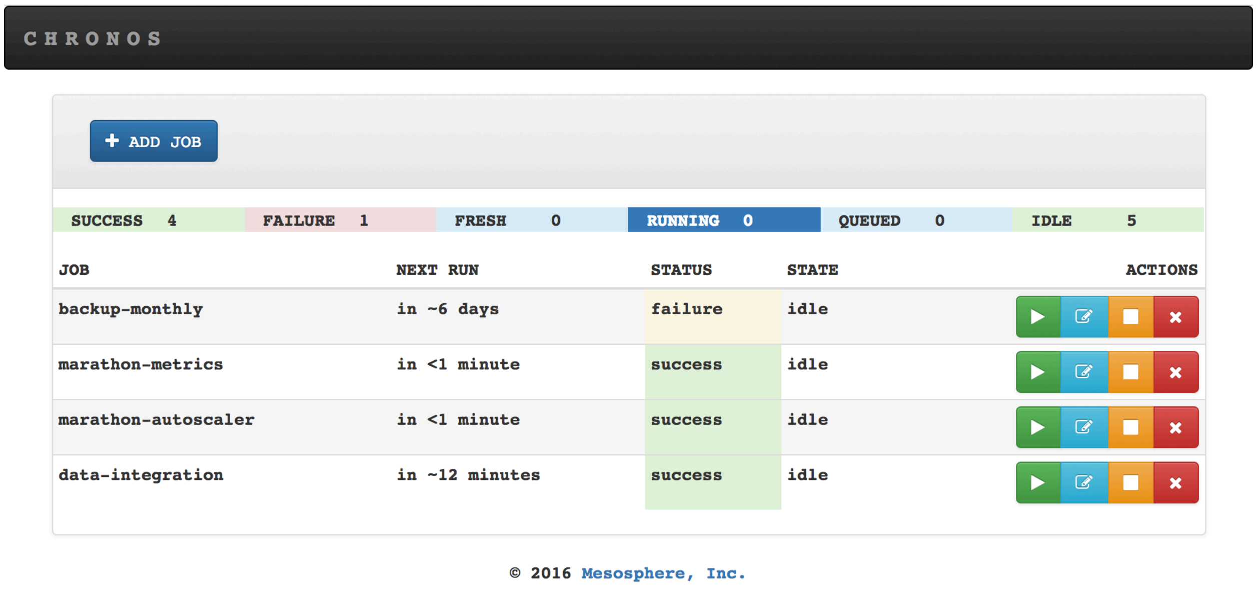
Synchronism: Wrong clock synchronization or misconfigured timezones might result in jobs being executed outside their expected times, which might introduce unexpected results.

Reactiveness: This solution does not address reactive job execution.

A.4.8 *Related Patterns.* Being an extension to ORCHESTRATION MANAGER, choosing a JOB SCHEDULER implementation typically is aligned with the ORCHESTRATION MANAGER choice.

Google also describes how to reliably schedule jobs using their cloud [Google 2018]. Using the Chronos JOB SCHEDULER on top of an Apache Mesos ORCHESTRATION MANAGER is explicitly described, as the authors have also done in subsubsection A.4.9.

Microsoft describes the behavior for a scheduler pattern [Microsoft 2017b], but it only explains how to implement one. This pattern follows a different approach, detailing on how to use a JOB SCHEDULER with an ORCHESTRATION MANAGER rather then implementing one from scratch.


CHRONOS

+ ADD JOB

SUCCESS	4	FAILURE	1	FRESH	0	RUNNING	0	QUEUED	0	IDLE	5

JOB	NEXT RUN	STATUS	STATE	ACTIONS
backup-monthly	in ~6 days	failure	idle	▶ ✎ ■ ✖
marathon-metrics	in <1 minute	success	idle	▶ ✎ ■ ✖
marathon-autoscaler	in <1 minute	success	idle	▶ ✎ ■ ✖
data-integration	in ~12 minutes	success	idle	▶ ✎ ■ ✖

© 2016 Mesosphere, Inc.

Fig. 9: The Chronos configuration user interface, showing four scheduled jobs. Chronos enables job scheduling on top of Mesos using a graphical user interface.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from Kubernetes
          restartPolicy: OnFailure
```

Listing 5: Kubernetes configuration for scheduling the execution of a container every minute.

A.4.9 *Known Uses.* Most infrastructure management environments have a companion scheduler service, either bundled in or as a plug-in service.

Chronos is a distributed and fault-tolerant scheduler for the Apache Mesos framework [Chronos 2017]. It exposes an API and user interface with which jobs can be scheduled and monitored. Figure 9 shows the Chronos user interface, with four jobs configured. Their state and recurrence is easily perceived in the status and state column, respectively.

Kubernetes enables job scheduling by making available a built in scheduler service. Similar to Chronos, jobs can be managed using the user interface or API [Kubernetes 2017]. Kubernetes API uses the YAML [4] format to describe jobs, as demonstrated in Listing 5.

Without using a Orchestration Manager, but with a similar objective, cloud providers tend to provide their own implementation of a scheduler, which can be used to manipulate their environment or client applications directly [Amazon 2017b; Microsoft 2017b]. These typically enable calling the provider's API to start some action such as running an anonymous function or starting a virtual machine or container.

It was also observed that some companies use a scheduler to periodically evaluate the infrastructure's load and appropriately resize it to cope with the current incoming traffic.

A.4.10 *Further Consideration.* Most Job Scheduler implementation respect the syntax specified by the POSIX utility Cron, as represented in Figure 10 [IEEE and Open 2016], for scheduling jobs. This syntax, despite not being a standard, has since been widely adopted as the de facto syntax for describing recurrent jobs, as seen in subsubsection A.4.9.

While scheduling is an common approach to schedule one time and recurrent jobs, there is another approach to it. The event-driven community [Fowler 2017] defends that a reactive is the most efficient way to identify when jobs should be spawned [Bonér et al. 2014]. With this approach, a Job Scheduler would not be needed, but an additional component to register event subscription could be adopted, defining which jobs should be spawned after a specific event if observed. For the specific case of time-based execution, this component could react to the clock ticks.

[4]YAML is a human friendly data serialization standard for all programming languages. Learn more at `http://www.yaml.org/`.

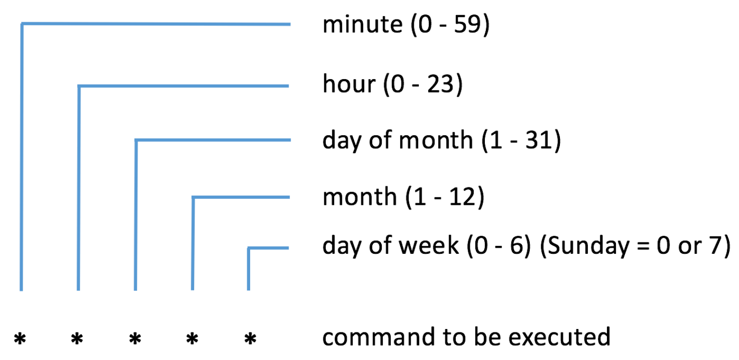

minute (0 - 59)
hour (0 - 23)
day of month (1 - 31)
month (1 - 12)
day of week (0 - 6) (Sunday = 0 or 7)
* * * * * command to be executed

Fig. 10: Overview of the CRON format, a commonly adopted syntax used to specify the date and time at which a job should be executed and repeated.

A.5 Failure Injection

Resilience mechanisms are triggered when software is failing. Since systems are designed to work correctly, the status quo prevents us to from continuously verifying the correctness of those mechanisms. We need additional strategies to minimize the probability of failure in production due to faulty resilience strategies. FAILURE INJECTION software can generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms, verifying the application's resilience

A.5.1 *Context.* Software fails [Charette 2005]. This assertion is widely accepted and the motivation for writing resilient software. Application failures can originate both from malfunctioning software or due to external conditions, which might be impossible to predict, such as network failures or defective hardware.

When running software at scale, issues are statistically guaranteed to happen [Pinheiro et al. 2007]. As such, it is imperative that cloud software is designed with resilience in mind, meaning that the application should have a set of strategies to recover from problematic situations at both the application and infrastructure layers. Still, resilience strategies are themselves software, hence, prone to failure, limiting the confidence on their efficiency.

A.5.2 *Example.* Consider a online web application powered by a database. Such database is essential for the system to work. As such, the database is replicated in hot-standby mode, meaning that the second instance has a complete copy of the first, being used for failover. Furthermore, the database is frequently backed up to an off-site using AWS S3 and Azure disk snapshots.

Consider now that the second database has an issue and needs to be manually resynchronized. While doing so, by mistake, an operator manually deletes part of the production database, leaving both inconsistent and loosing production data. When trying to recover the database from the off-site backups, the operator identifies that the backups are not available and identifies that the backup procedure has not been running as expected. No recent backup is available and the operator will not be able to recover the database to a recent state, resulting in the actual loss of production data.

The example above is actually a simplified version of an event from early 2017, when a GitLab engineer accidentally deleted part of their production database, only to understand that the existing recovery mechanisms where not properly configured, leaving the system down for over 18 hours and resulting in the actual loss of production data, namely in the changes to projects, comments, user accounts, issues and snippets, that took place between 17:20 and 00:00 UTC on January 31 [Gitlab 2017].

A.5.3 *Problem. Resilience mechanisms are triggered when software is failing. Since systems are designed to work correctly, the status quo prevents us to from continuously verifying the correctness of those mechanisms. We need additional strategies to minimize the probability of failure in production due to faulty resilience strategies.*

It has been previously asserted that software fails. That was the main motivation behind the *let it crash* philosophy in the Erlang language and other actor models, where instead of defensively addressing all possible errors, the program was allowed to crash and restarted in an attempt to recover normal execution [Cunningham 2014]. The Reactive Manifesto also addresses this type of recovery, with resilience through recovery as being one of the four characteristics of reactive systems [Bonér et al. 2014].

By relying on software as a recovery mechanism for other software, it is possible that the recovery mechanisms might fail as well. For that matter, just like any other application, the recovery mechanisms themselves must be validated and frequently tested to ensure their correct behavior.

While designing resilience processes for cloud software, these processes themselves should be monitored, ensuring that the system is able to properly recover from from failure.

Verifying resilience presents the same problem as verifying software: it is not possible to guarantee that the system is completely resilient, only that it endures the identified test scenarios. Furthermore, testing software for bugs is easier than testing resilience, as resilience might be influenced by the underlying infrastructure that hosts the application, which might not be under them team's control. As such, resilience testing is not an one time activity, but instead needs to be continuously improved during the lifetime of the application.

At its core, verifying resilience requires the implemented processes to be stressed, putting the application through unexpected scenarios and verifying how well it behaves. This might be problematic by itself if at some point the application is unable to recover without manual intervention, rendering it in a degraded state.

This problem becomes further complex as it is insufficient to verify resilience in a staging environment, given that resilience is highly influenced by a multitude variables in the infrastructure, such as number of resources allocated, for how long they have been allocated or how much load they are handling. While it is possible to create a similar staging environment, even the specific hardware allocated to production might present a different behavior from the staging environment. As such, the only way to increase trust over the resilience of a production environment is to actually test that environment.

A.5.4 *Forces.* The following forces, represented in Figure 11, need to be balanced while considering the adoption of this pattern:

Preemptive failure detection: Identify failures in the application before they accidentally impact the application or are exploited by third parties.

Failure generation: Known failures are less likely to cause impact in the system than artificially generated ones.

Resilience: Failure injection might degrade the status of the system.

A.5.5 *Solution. Generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms to verify the application's resilience*

To ensure that the system will recover when a problem arises, its resilience strategies must be frequently exercised, even in production, ensuring that the system does in fact recover to the expected status when a failure happens.

An external piece of software can frequently generate unexpected events at both the application and infrastructure level and monitor how the system behaves, verifying it if it recovers as expected. These events can range from shutting down a container instance to a full virtual machine server. In both scenarios, the resilience strategies should be able to recover the application to the expected status, restarting the container in the first scenario or the machine and it's hosted services in the second.

The adopted strategy used for FAILURE INJECTION should be aware of the application and infrastructure's APIs and randomly inject invalid payloads or shutdown system components.

While implementing this pattern, one must consider:

Completeness: failures can be injected at both the infrastructure and application level. Only by testing both can we maximize the level of confidence in the system's resiliency.

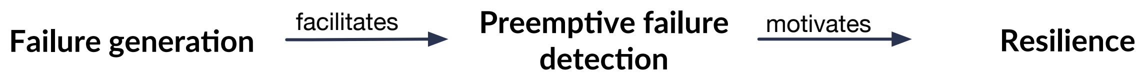


Fig. 11: Relationships between FAILURE INJECTION force.

Frequency: We need to decide how often we will exercise the resilience mechanisms, balancing how much resources we are willing to allocate, which directly impacts execution cost, with the level of trust we want to continuously have over the system.

Traceability: We need to understand the impact of injecting failures in the system, by aggregating information from the failure injection system with the infrastructure and application logs, facilitating the evaluation of the impact of a failure injection on the system.

Programatic failure injection: We want to enable the developer to programatically describe his failures or failure generation logic, so that the failure injection can be automated and executed automatically, reducing the need for manual intervention while running failure injection tests.

There's a several attack vectors and liabilities introduced while testing resilience. The following scenarios should be considered:

Application misuse: Generate random inputs to the application's interfaces, including its APIs.

Unexpected load: Suddenly increase the system's load, by generating an abnormally high amount of traffic.

Network degradation: Degrade or disable the network to a server, either by disabling the server's network card or use an application that consumes its bandwidth.

Resource depletion: Deplete available disk, RAM or CPU from a server, by starting an application in the server that consumes such resources.

Unexpected component shutdown: Shutting down random servers or other system components, up to disabling entire availability regions.

While exercising the recovery mechanisms with FAILURE INJECTION, the system is expected to be impacted, which should be carefully monitored by observing:

Latency: Some tests will degrade or shutdown resources. While doing so, application latency should be monitored. An ideal resilience mechanism will recover from the injected time without increasing latency above the expected limit. Data from the EXTERNAL MONITORING pattern can be leveraged to observe the application's latency from the user's perspective.

Recovery time: The application should recover within an expected duration. Infrastructural and application logs can be used to verify if a recovery is taking more time than expected, which will introduce the need to improve the resilience mechanisms.

Data: A resilient application should be able to recover from a failure without losing or corrupting data.

Security: During the recovery of the application, the system should remain secure, ensuring that no temporary attack vector is introduced.

Supervision and monitoring patterns such as EXTERNAL MONITORING are companions to FAILURE INJECTION. It is expected that some of the generated events will degrade the system, but its resilience should enable automatic recovery, preventing any impact on the application. If such doesn't happen, monitoring patterns should identify the degraded system state, providing the required information for the development team to recover the system and afterwards implement the required steps to improve its resilience.

It is arguable if FAILURE INJECTION should be applied to production environments given the risk to degrade them. To prevent impacting production systems, FAILURE INJECTION should first be thoroughly tested in a development or staging environment, being introduced into production when the level of confidence around the application's resilience if definitive. Furthermore, its execution in production environment should be constrained to work hours, under close supervision of the team.

While it is arguable if FAILURE INJECTION should be executed against production systems, exercising its recovery mechanisms is the only way to ensure that they are working properly.

A.5.6 *Example Resolved.* Adopt a FAILURE INJECTION tool and configure it to generate failures against the application's database and its infrastructure, insuring that the system is able to recover automatically.

By periodically exercising the database reliability, the team would have been able to identify earlier that the backup process was not working, just as well as it would be able to understand that the hot-standby replication was not optimally configured, improving it to be sure that the secondary server would be able to sustain the expected level of service required by the application.

A.5.7 *Resulting Context.* This pattern introduces the following benefits:

Automated Failure detection: The adopted tool will generate and inject random events in the system, testing it thoroughly and continuously, identifying issues faster than any manual testing could.

Awareness: Using the EXTERNAL MONITORING pattern, the team can be notified of a degradation whenever a FAILURE INJECTION impacts the system.

Preemptive failure detection: By stressing the application with unexpected events, the team is able to preemptively identify failures that could happen in the wild otherwise.

On the other hand, the following limitations will be introduced:

Availability: While testing reliability, it might be the case that an issue is identified and the system's performance degraded. The team should be immediately alerted and take the required actions to recover the system's stability, as well as implementing the required automations to recover from the newly identified scenario.

Resource usage: Exercising resilience will only be possible when resilience mechanisms are available. Often resilience requires redundancy to be implemented, which will always increase the resources required to operate the application.

Unintended consequences: While the system might be able to recover, it might do so while introducing unacceptable consequences. For example, a critical system might lose data during a recovery process.

A.5.8 *Related Patterns.* When implementing this pattern, SELF HEALING should have been implemented, enabling both the application and infrastructure to recover automatically. FAILURE INJECTION can also leverage LOG AGGREGATION for capturing its action.

The description of the responsibilities for a FAILURE INJECTION tool has been described the Software Failure Injection Pattern System [Leme et al. 2001].

A.5.9 *Known Uses.* Netflix was one of the main motivators behind FAILURE INJECTION with the implementation of their open source tool ChaosMonkey. ChaosMonkey interacts with an AWS account and randomly shutting down infrastructure components. At Netflix, ChaosMonkey is executed against the production environment during business hours, randomly terminating virtual machines. Their rationale is that exposing engineers to failures motivates them to make their services more resilient [Netflix 2017]. ChaosMonkey is one of the many tools available in the Simian Army, a set of open source tools developed by Netflix to help engineers improve their software's reliability[Netflix 2011].

Motivated by the impact from the floods of Hurricane Sandy in 2012 in New Jersey, Project Storm is Facebook's approach to resilience testing. At its infancy, it was composed by a set of small drills lead by a reliability team that were designed to replicate the consequences of catastrophic natural events, just like Hurricane Sandy was, by degrading or disconnecting small parts of their infrastructure. By 2014, the team behind Project Storm upped their game, starting to disable entire data centers. The initial drills enabled the team to identify several unexpected points of failures [Hof 2016].

FAILURE INJECTION is motivated by the Principles of Chaos Engineering. Quoting, "Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production" [Chaos Community 2017]. In practice, it consists on experimenting with the moving parts of the application, looking for actions that might result in a system failure, such as crashing servers of malfunctioning hard drives.

A.5.10 *Further Considerations.* Chaos engineering practices are implemented against systems expected to be reliable, validating their reliability. It should be expected that failures are found and the system should recover without manual intervention. Still, for teams starting to implement FAILURE INJECTION, its execution should be carefully monitored, as some failures might result in non considered scenarios, leaving the system in an unrecoverable state and requiring manual intervention.

According to the Chaos Community, Chaos Engineer is based on the following principles [Chaos Community 2017].:

Build a Hypothesis around Steady State Behavior: Focus on the measurable output of a system, rather than internal attributes of the system. Measurements of that output over a short period of time constitute a proxy for the system's steady state. The overall system's throughput, error rates, latency percentiles, etcċould all be metrics of interest representing steady state behavior. By focusing on systemic behavior patterns during experiments, Chaos verifies that the system does work, rather than trying to validate how it works.

Vary Real-world Events: Chaos variables reflect real-world events. Prioritize events either by potential impact or estimated frequency. Consider events that correspond to hardware failures like servers dying, software failures like malformed responses, and non-failure events like a spike in traffic or a scaling event. Any event capable of disrupting steady state is a potential variable in a Chaos experiment.

Run Experiments in Production: Systems behave differently depending on environment and traffic patterns. Since the behavior of utilization can change at any time, sampling real traffic is the only way to reliably capture the request path. To guarantee both authenticity of the way in which the system is exercised and relevance to the current deployed system, Chaos strongly prefers to experiment directly on production traffic.

Automate Experiments to Run Continuously: Running experiments manually is labor-intensive and ultimately unsustainable. Automate experiments and run them continuously. Chaos Engineering builds automation into the system to drive both orchestration and analysis.

Minimize Blast Radius: Experimenting in production has the potential to cause unnecessary customer pain. While there must be an allowance for some short-term negative impact, it is the responsibility and obligation of the Chaos Engineer to ensure the fallout from experiments are minimized and contained.

A.6 Preemptive Logging

The information required to debug issues in software is often lost during their first occurrence due to insufficient log verbosity. By adjusting logging verbosity preemptively in services and servers within acceptable resource limits (CPU, storage, others), the team maximizes the probability of capturing relevant information for addressing future issues right from their first occurrence.

A.6.1 *Context.* It's often difficult to guarantee that software will behave as expected, but it can be designed for the worst. In those situations, information is key to debug applications, which makes having execution logs available after those unexpected scenarios the most relevant piece of information to understand what, how and why the software has failed.

Most third-party applications have adjustable verbosity logging capabilities, but first-party applications sometimes neglect that need, causing the developers to lack the required information to mitigate unexpected failures. Given that service cooperation is key in cloud software, and considering the uncertainty of the events that lead to unexpected errors, all services should equally generate logs that are the sole resource from developers to understand and mitigate the issue.

A.6.2 *Example.* Consider a database service in a microservice architecture. The service is responsible for persisting information important for other services in the infrastructure. At a given point in time, the database crashes. Automated operations practices should ensure that the service is automatically recovered, but, after a while, it crashes again. This behavior is recurrent and without explanation from the development team. The team is expected to identify and fix the issue, but is not being able to reproduce it outside the production environment. Without the proper information about the production system, the team is rendered incapable to properly addressing the issue.

A.6.3 *Problem. The information required to debug issues in software is often lost during their first occurrence due to insufficient log verbosity.*

Development teams tend to be conservative on their software instrumentation, undervaluing the importance of capturing runtime information. When software fails, it is common that the only debugging approach is to further instrumenting the software and await for the issue to repeat itself. This approach decreases the level of confidence on the software quality, as well as requires the team to knowingly leave a bug in their software given the lack of information to fix it.

A.6.4 *Forces.* The following forces, represented in Figure 12, need to be balanced while considering the adoption of this pattern:

Traceability: Development teams need as much data as possible to be available in order to identify the conditions that may have triggered issues in a service.

Execution Resources: Increasing the logging level increases the amount of resources required to execute the service, such as CPU and memory.

Retention Policy: Verbose logging can become expensive to collect and persist for large periods of time.

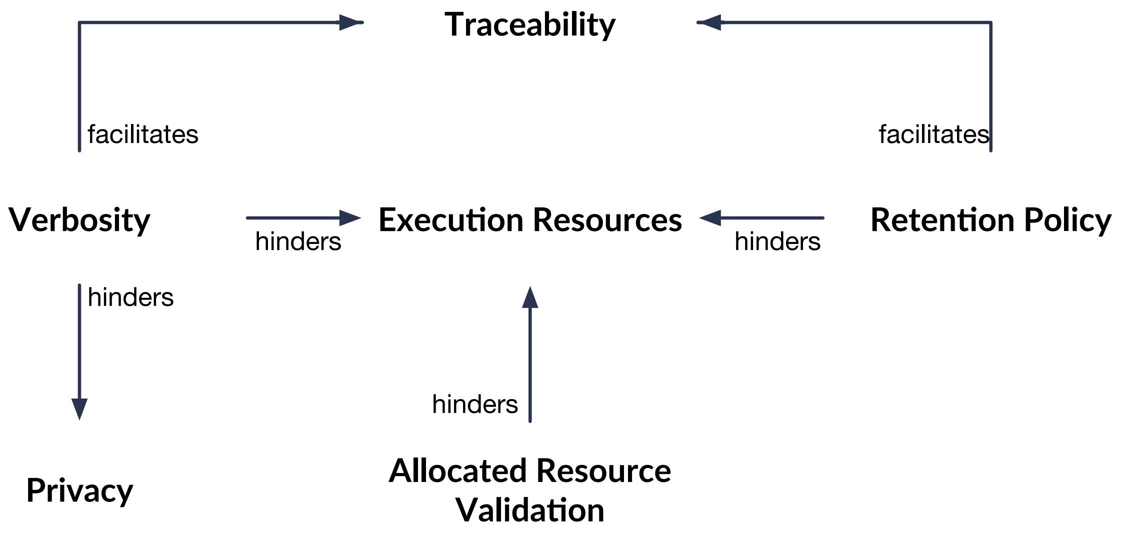


Fig. 12: Relationship between Preemptive Logging forces.

Verbosity: Increasing the log verbosity provides additional information for posterior debug, but it also requires additional storage space and human effort to process.

Allocated Resource Validation: Resources might be over or under-allocated to a service, result in a poor usage of the available infrastructure resources.

Privacy: Due to legislation, it might not be possible to persist some type of data.

A.6.5 *Solution. Adjust logging verbosity in services and servers within acceptable resource limits (CPU, storage, others), maximizing the probability of capturing relevant information for addressing future issues right from their first occurrence.*

Logging is often undervalued by less experienced developers, who are tempted to reduce log verbosity in production to keep the system leaner. By doing so, they unintentionally miss the opportunity of capturing information that would allow them to debug unexpected runtime problems. This may prevent the development team from effectively tackle such problems, unless they begin monitoring the service and server, hoping to observe the issue happening again and capture enough information to identify the reasons behind it.

PREEMPTIVE LOGGING insures that runtime information from both services and servers is captured, being an asset for debugging runtime issues.

To implement it, development teams should start by discussing and identifying all the information that can possible be extracted from the service and respective server. From there, the team should discard the items that will never be useful, setting the optimal log verbosity level for them.

While deciding on what data to keep, resource usage should be discussed, has the more information is persisted, the greater the resource impact in the system. A retention policy should also be set, as the information becomes less relevant with time.

A scenario where all system events can be captured is ideal, as these can be reproduced in a test environment to further debug issues. Also, once a bug is fixed, they can be replayed in the production environment some types of failures, e.g. one where a specific service is dropping the events sent to it.

Recent privacy trends such as the European GDPR regulation might prevent some event data to be physically persisted.

A.6.6 *Example Resolved.* The team responsible for the database service would discuss what metrics would be relevant to understand how the service is behaving. As an example, they could capture: number of incoming connections, number of incoming queries, query response times, programming exceptions and the incoming queries themselves. Server metrics would also be captured, namely disk IO, RAM, CPU or network usage.

If the service revealed an issue, they would access the generated log files and use them to understand what triggered it. They could start by understanding if the allocated resources where enough to accommodate the service. If that hypothesis is excluded, they could then dig into the service's own logs in order to understand when and why the service started to misbehave.

A retention policy can automatically archive or delete older log entries. When adjusting this policy the team should allow enough time to ensure the logs are available during the time period when they might be used, preventing them to get discarded too soon.

A.6.7 *Resulting Context.* By adopting PREEMPTIVE LOGGING, development teams will gain:

Reproducibility: Service operations can be captured, facilitating the team to understand how it has behaved. The whole input stream can be captured and replicated in a controlled environment to understand how and why it reacted in a certain way to a given set of inputs.

Allocated Resource Validation: Capturing logs from hardware usage from a server will enable the team to better understand how the service consumes resources and optimize resource allocation.

Security and Auditing: The development team will be able to trace security problems and threads.

While configuring the service's logging levels, the following should be taken into consideration:

Resources: Should be increased to cope with both the higher CPU and disk space demand of increasing the log verbosity.

Retention Policy: An increased retention policy will keep the logs available for a long time period, but such will increase the required disk space required to persist the logs.

Verbosity: Again, the verbosity level should be adjusted to a value that balances a relevant output, with the amount of disk space it will consume.

Security: An attack on this component would expose information from all others.

A.6.8 *Related Patterns.* The team job's is simplified if it is able query log entries from multiple sources, understand what events where happening in each service. LOG AGGREGATION provides this functionality, by moving the logs from their origin to a centralized repository, where they are aggregated and indexed, facilitating their usage. COLLABORATIVE MONITORING AND LOGGING describes the importance of logging and its relevance while deployment software on the cloud [Arcitura Education Inc 2017]. Fernandez described how logs can be leveraged to audit security in the AUDIT LOG pattern [Fernandez 2013].

A.6.9 *Known Uses.* Amazon Web Services' CloudTrail enables the capturing of all API interactions in an AWS account, providing complete traceability of all changes through it [Amazon 2017a]. Azure provides a similar service [Azure 2017]. Spinellis identified log verbosity as a parameter to manually tweak in production when looking for problems [Fu et al. 2014]. Fu elaborated on that problematic in his survey [Fu et al. 2014], theorizing automated log verbosity adjustment in production as a relevant research topic.

A.7 Log Aggregation

Services orchestrated at scale produce disperse logs, resulting in a troublesome process to acquire and correlate those who come from multiple sources. This pattern suggests the Aggregation and indexing all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs.

A.7.1 *Context.* At the scale at which cloud Computing is applied, and given the scalability of operations introduced by DevOps, a development team can easily be managing hundreds of services orchestrated on top of thousands servers. Both the hardware and their hosted services are continuously operating and producing relevant information, commonly via log files. Those files must be accessed often and it is not functional to keep them dispersed in the infrastructure, forcing developers to individual access each machine and file to gather the information they require.

A.7.2 *Example.* Consider the example from section A.9.2, where each service is running on its own dedicated server. The three services are producing log files, along with the operative system from their host. Imagine now that there was an issue with the AC service or server, rendering the service unresponsive. The developers need to remotely login to the server to access the required log file and debug the issue. Along this process, they understand that the issue was due to a communication error with the messaging service. They now need to access the machine hosting the messaging service in order to debug its log entries. This process must be repeated for each service and server involved in the issue, going back and forth until the problem is identified. This approach makes it difficult to correlate log entries from different sources and demands that the developer individually access each one of the machines.

A.7.3 *Problem. Services orchestrated at scale produce disperse logs, resulting in a troublesome process to acquire and correlate those who come from multiple sources.*

Teams deploying software at scale can easily see their infrastructure grow to tens of servers hosting hundreds of service instances. As suggested by PREEMPTIVE LOGGING, these services should be verbose at producing logs. At this scale, it is troublesome for developers to leverage this logs, given their sparsity across the infrastructure. The basic solution of individually accessing each server and service log file quickly becomes unmanageable.

A.7.4 *Forces.* The following forces, represented in Figure 13, need to be balanced while considering the adoption of this pattern:

Fragmentation: Scattered log files across servers incur in extra effort for the developers to debug the application.

Network Propagation: Transferring log data from its source to a central aggregation point requires additional bandwidth and might incur in additional data transfer costs.

Ordering: Propagation of logs through the Internet and unaligned clocks might result in out of order log entries.

Querying: Querying in a log stream is essential to quickly identify relevant information from large collections of logs.

Security: Sending logs across the network should use a secure channel, insuring that sensitive information is never stolen. Also, the log storage should guarantee that they are not writable, preventing attackers or other software from changing them.

A.7.5 *Solution. Aggregate and index all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs.*

Having logs available only at their source makes their usage troublesome, requiring the user to login into the system and either download them or use the set of tools remotely available to query them. At scale, when tracing how services cooperate with each other, this means that the user would have to replicate this process across all intervening servers and services.

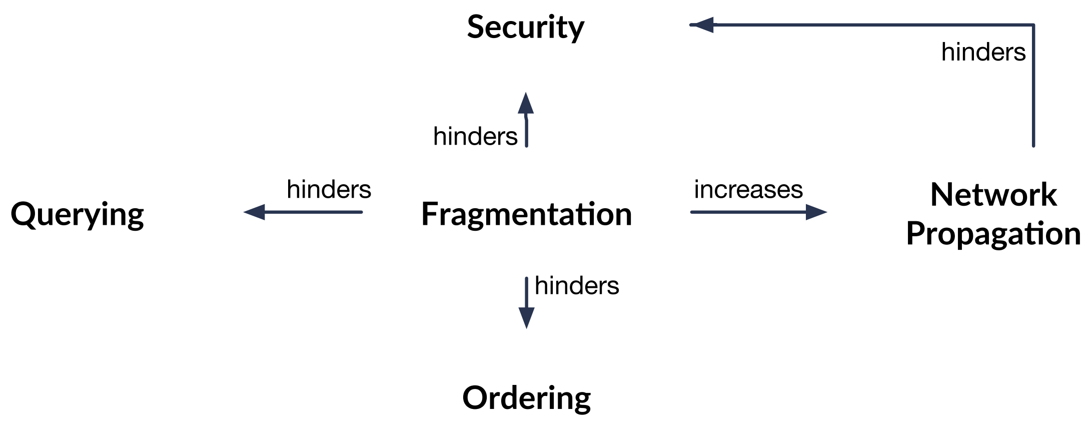


Fig. 13: Relationship between LOG AGGREGATION forces.

LOG AGGREGATION solves this problem by providing a centralized system for aggregating and visualizing all logs in an infrastructure. This solution is applied as: (1) a log aggregation service is deployed in the infrastructure, enabling the querying and visualization of information from the logs and (2) each service daemon deployed along with it must forward its logs to the log aggregation service.

The centralized log service can persist the log entries in a database, exposing a query interface for them. This allows developers to mix and match events, better understanding what has happened in their infrastructure as a whole, or in a specific machine or server. Figure 14 represents the involved components in this process as a class diagram.

A secure channel should be used when sending logs from their origin to this centralized database. Also it should allow entries from being written, but prevent them from being changed, ensuring that logs are immutable.

A.7.6 *Example Resolved.* Each service and server would send their logs to a centralized log repository service. This service would need to be instantiated in the infrastructure or adopted as an external service. Within it, the developer would have a global view of all logs from all services in the infrastructure. It would be possible to query those logs, filtering them specifically for any specific service at any given time.

A.7.7 *Resulting Context.* This pattern introduces the following benefits:

Fragmentation: Developers can use a aggregation service to aggregate all the information they need from any service or server in the infrastructure.

Querying: Once aggregated in a single location, data can be indexed, allowing developers to query the logs, finding the information they need for their specific task faster.

Security: Communicating logs using a secure channel is essential for keeping sensitive data private. Also, the chosen log storage should also be secured to prevent data leakage.

While deciding the technologies to implement this pattern, the following should be taken into consideration:

Network Propagation: In order to propagate logs to the aggregating server, additional bandwidth will be consumed.

Ordering: Ordering will rely on the time stamp generated at the server. There might be some errors in cases where the server's clock isn't synchronized.

Single point of failure: Without a redundant deployment, a failure in log aggregation system would revert this system's benefits.

A.7.8 *Related Patterns.* REPOSITORY describes a generic approach to a data repository [Hieatt and Mee]. Fernandez describes the application of log aggregation in the security context to trace user actions [Fernandez-Buglioni 2013].

MESSAGING SYSTEM can be used as a communication channel to propagate logs to the log the aggregation service. This pattern is further useful if PREEMPTIVE LOGGING is applied in each service in the infrastructure.

LOG AGGREGATION can be used as a source of information for REACT, feeding it with the events used to trigger reactive actions.

A.7.9 *Known Uses.* Elastic, through their Elastic Stack, leverage Logstash as a tool for acquiring and propagating logs from applications. Logs are propagated to a remote Elasticsearch, an highly indexable JSON document storage. Information can then be queried and visualized using Kibana, a dash-boarding tool for captured data [Elastic 2017].

Loggly[5] is a subscription based log aggregation cloud service. It provides clients for acquiring logs from multiple platforms and services, making them available in a single time-based searchable history.

[5]Details at `https://www.loggly.com/`.

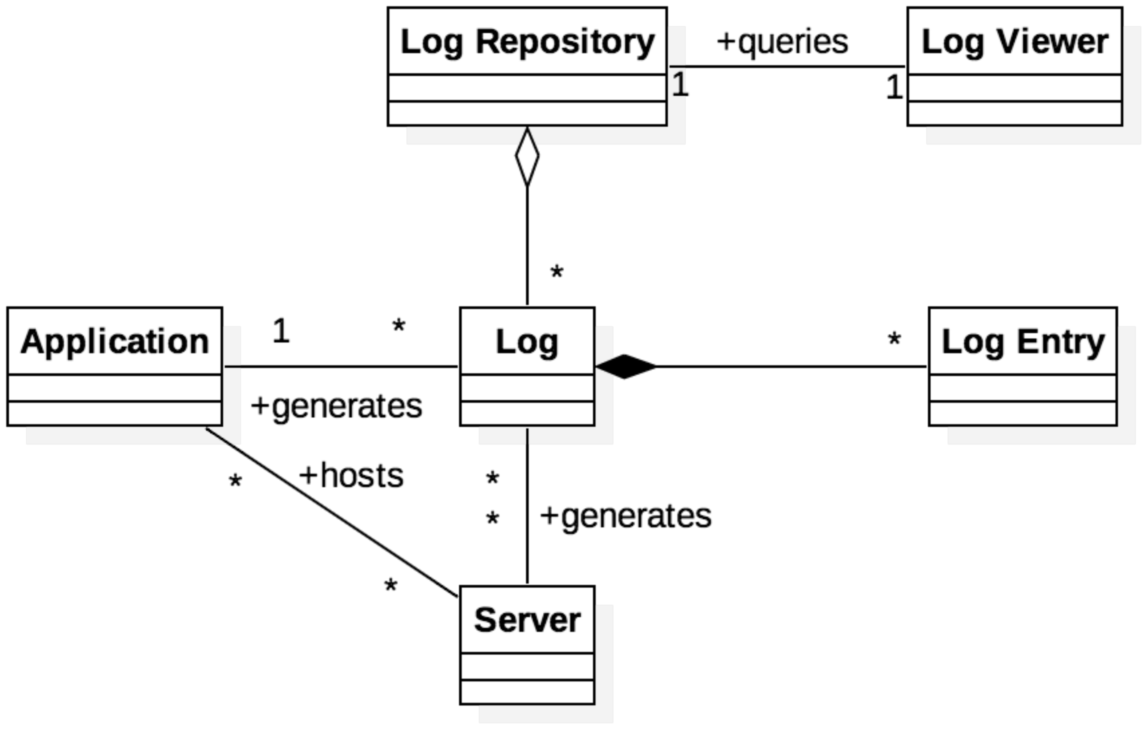


Fig. 14: Class diagram showing the entities involved in the log generation, persistence and querying process. Applications and Servers generate log files which are composed by multiple log entries. Each log entry as relevant information of the team to use in the future. Logs are persisted in a remote log repository. This repository aggregates all sources of information, allowing the log viewer to query a single location.

Roderick et all have described how their logging service acquired over 50 TB per year, making this data available for over 1000 users daily [Roderick et al. 2013].

A.8 External Monitoring

Monitoring an application from inside the infrastructure that hosts it will result in an incomplete and biased version of the reality, for example, given the inability to observe issues such as lack of Internet connectivity or abnormal latency to the application. EXTERNAL MONITORING suggest testing the application's public interfaces from an external source, providing an unbiased awareness of the application's status.

A.8.1 *Context.* While part of the development process is responsible to ensure resilience, just like it is impossible to ensure complete reliability using software testing [Malaiya et al. 2002], it is not possible to ensure that a system is 100% resilient. Accepting that it will eventually fail is important to accept the need to increase awareness about the system's status at all times. This is the motivation behind the adoption of monitoring systems.

Frequently, monitoring systems live within the application's own infrastructure, which might bias the awareness about the actual state of the application, given all the external variables introduced by using the Internet as a distribution channel.

A.8.2 *Example.* Consider an authentication service, part of a larger application. Provided with a valid login, it should output an authentication token for interacting with the other services in the application. Consider the scenario where when used from within the infrastructure, the authentication service works as expected, but, when accessed from a remote application, the authentication service is inaccessible. Such discrepancy might have been caused by a misconfigured firewall.

This scenario demonstrates that a service can have different status when observed from within the application's infrastructure and a remote site.

A.8.3 *Problem. Monitoring an application from inside the infrastructure that hosts it will result in an incomplete and biased version of the reality, for example, given the inability to observe issues such as lack of Internet connectivity or abnormal latency to the application.*

Software failures can be catastrophic to business owners. Application downtime consequences can range from client complaints to loss of confidence in the application and, ultimately, user abandonment or contractual breach. Given the ever growing offer of online services, a failing application can easily be replaced by a competitor.

In case of failure, the development team should quickly be aware of the application's status, facilitating a quick reaction.

This awareness must not depend on the application or its infrastructure, as that would bias the observation. In the context of cloud computing, simply monitoring the application alongside its execution is not only biased, but prevents the detection of several unpredictable Internet-related issues, such as misconfigured or failing routers, CDN, DNS or firewalls, which would directly impact the client's access to the application.

In the example from subsubsection A.8.2, a misconfigured firewall is inadvertently blocking traffic from a valid source, leaving the service inaccessible from the outside. This issue would not be identified by monitoring the application from within the infrastructure, as the firewall would not be used between two internal services.

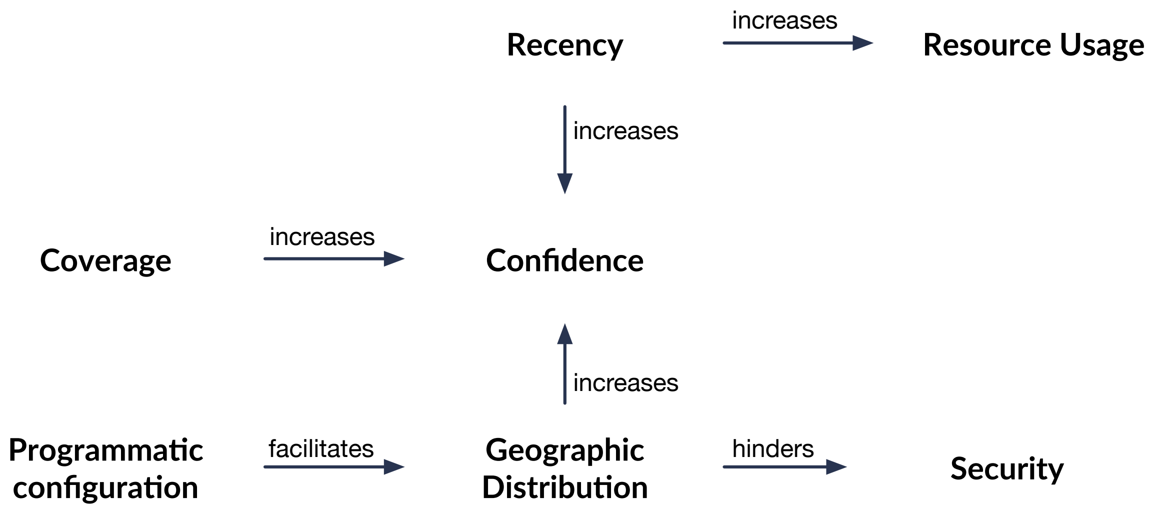


Fig. 15: External Monitor forces relationships.

A.8.4 *Forces.* The following forces, represented in Figure 15, need to be balanced while considering the adoption of this pattern:

Confidence: Maintain awareness of the system's state without relying on its internal information or be biased by internal monitoring.

Recency: Be notified as soon as a possible complication is identified in the application.

Coverage: Confidence level is increased with the increase of test coverage.

Resource usage: Minimize the impact from monitoring on the application's resource requirements, which will directly impact either performance or cost.

Security: Minimize the attack vectors for the application. Exposing sensitive application details to additional external tools will create a new attack vectors.

Geographic description: Running tests from different globe locations increases the level of confidence that the system is working worldwide.

A.8.5 *Solution. Test the application's public interfaces from an external source, providing an confidence over the application's status.*

Resilience is an essential requirement of any cloud software. Still, just like with software testing, it is impossible to guarantee that a system is fully resilient and that it will not fail. Besides improving the system's resilience, the development team should also invest in their awareness of the system's status in production, reducing the time required to detect a failure.

External Monitoring consists on the frequent execution of tests against the public interfaces of a live production system, evaluating if they are responding as expected. Tests are configured and executed from in a service running in a separate network environment from the application itself and run without any knowledge of the application's state (as a black-box test), providing an accurate observation of the system's status as seen from across the Internet.

Test coverage can range from a basic status check to see if the service is up to having a batch of tests covering all the application's public interfaces and their different uses. Such level of coverage could be seen as black-box integration tests executed against a live environment. It is up to the team to balance the level of coverage with the intended level con confidence in the system's status.

The team can either develop their own External Monitoring tool or adopt one of the many third party tools available. Developing a tool for external monitoring would require a considerable investment in development and operations. On the other hand, adopting a third party tool introduces a financial cost for using the service, as well as it widens the attack surface to the application, as sensible information such as user credentials need to be shared with the system. An hybrid approach could consider adopting an open-source tool for doing external monitoring, which will prevent sharing sensitive credentials with a third party while still requiring little investment in developing the software.

Some tests might require sensitive data to execute, such as user credentials. In case of an attack to the monitoring platform, this might hinder security, leaving those credentials exposed. Frequently rotating these credentials can help mitigate this issue.

While implementing this pattern, one must consider:

Recency: We need to decide how often we will run the external monitoring tests, balancing how fast do we want to know when an issue appears with the system as the load introduced by the tests will increase resource usage.

Development effort: We need to balance the completion of test coverage with the time required developing new tests.

Security: We need to decide which, if any, credentials should be made available in the external system to test protected interfaces, at the cost of possibly exposing sensitive data.

Accuracy: We want to prevent false positives by confirming issues redundantly, such as those who might result from latency or network partitioning.

Geographical distribution: We might want to distribute tests globally, ensuring the the application is working withing the specified parameters, despite where the traffic is originated. This enable verifying the correct behavior of components such as CDNs.

Traceability: We want to understand why a test has failed, by evaluating the inputs and outputs used to identify the failure. Log Aggregation pattern can be leveraged to combine logs from this patterns, as well as logs from Fault Injection and the remaining components of the system, providing an unified view over the system's behavior.

Programmatic configuration: We want to manage monitoring tests automatically as part of the deployment process, eliminating the need for manual configuration, hence, increasing confidence in the tests.

Third party tools for implementing the pattern often allow tests to be created from both a graphic interface, as seen in Figure 16 and a programmatic interface. The latter enables tests to be configured as part of the application's deployment process.

External monitoring is not a recent subject to cloud computing. *Cloud monitoring: A survey* [Aceto et al. 2013] thoroughly details why cloud computing is an important aspect of cloud applications and describes over twenty tools to implement it, ranging from commercial to open-source offers, being a good support for selecting the tool used to implement this pattern.

A.8.6 *Example Resolved.* Considering the example, this pattern would be implemented by adopting an External Monitoring system which would make an authentication request to the authentication system and confirm that the answer contained a proper authentication token. This test would be configured in the monitoring platform at the end of the deployment process, ensuring that the application is tested and working as expected right from the moment the deployment is complete.

Tests would be executed at a configured frequency and from different geographic locations to ensure that the application behaves correctly, despite from where a request has originated.

Possibly at a later stage, and for increasing test coverage, any other interface in the service could be tested as well.

To prevent configuring the external platform with actual user credentials, a mock user could be set up in the live system. This way, in the case of data leakage in the monitoring system, no significant impact would be observed on the monitored application. It is arguable that testing against a single user account that was created with the sole purpose of interacting with tests might bias the test results.

A.8.7 *Resulting Context.* By adopting External Monitoring, development teams will gain:

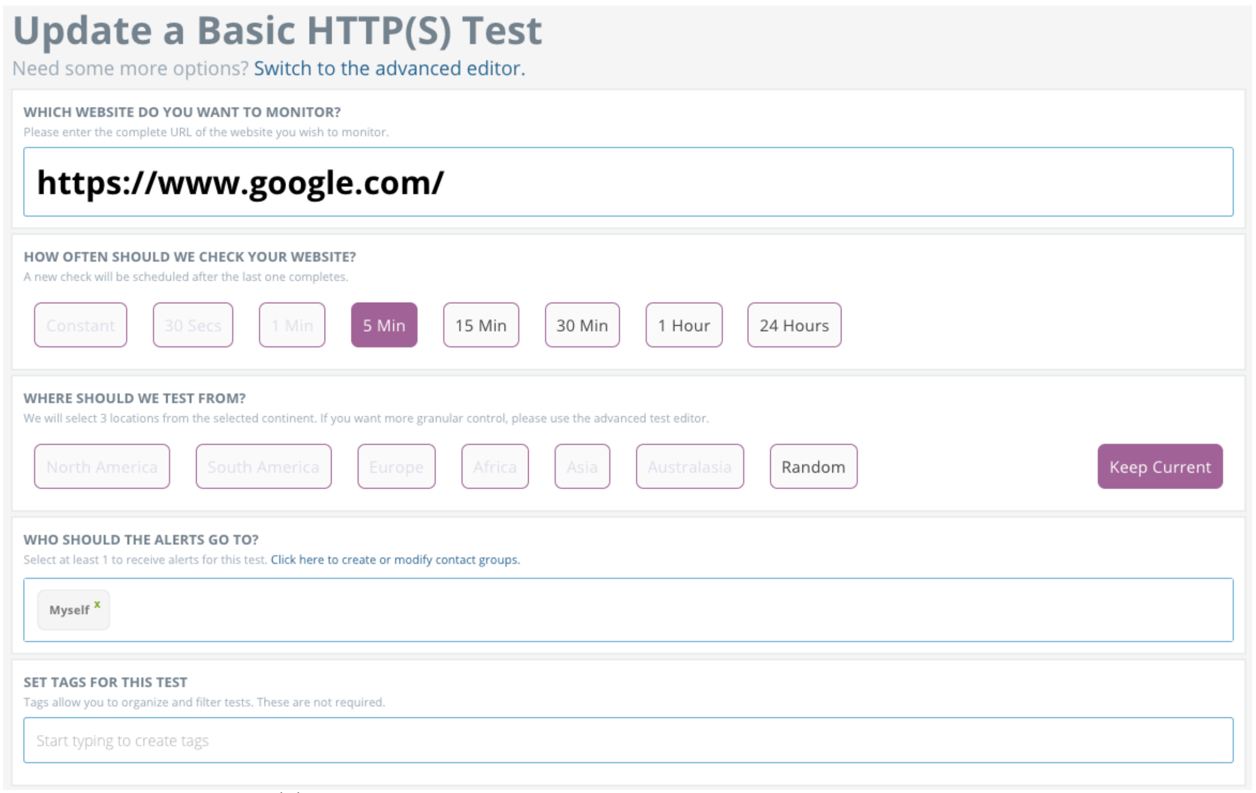
Confidence: Given continuous independent monitoring, there is an added confidence that the system is behaving as expected if no alarm is raised.

Traceability: The team will be able to understand what behavior was observed as response to any failing request using the external monitoring logs.

Programmatic configuration: The team will be able to evolve test scenarios along with their development, using the external monitoring API to setup or update tests.

On the other hand, the following liabilities can be introduced:

Security: When the communication channel is properly secure, no data leakage can occur by executing the tests from an external monitoring provider. The team must trust the provider though. Given an attack


Update a Basic HTTP(S) Test

Need some more options? Switch to the advanced editor.

WHICH WEBSITE DO YOU WANT TO MONITOR?
Please enter the complete URL of the website you wish to monitor.

https://www.google.com/

HOW OFTEN SHOULD WE CHECK YOUR WEBSITE?
A new check will be scheduled after the last one completes.

Constant | 30 Secs | 1 Min | [✓ 5 Min] | 15 Min | 30 Min | 1 Hour | 24 Hours

WHERE SHOULD WE TEST FROM?
We will select 3 locations from the selected continent. If you want more granular control, please use the advanced test editor.

North America | South America | Europe | Africa | Asia | Australasia | Random | [✓ Keep Current]

WHO SHOULD THE ALERTS GO TO?
Select at least 1 to receive alerts for this test. Click here to create or modify contact groups.

Myself

SET TAGS FOR THIS TEST
Tags allow you to organize and filter tests. These are not required.

Start typing to create tags

Fig. 16: Statuscake's HTTP(S) test creation interface, showing a basic HTTP test for Google's homepage, which will execute every 5 minutes from a random server.

against it, sensible information might be exposed. It is the team's responsibility to minimize or eliminate the need for sensible information such as credentials for executing the tests.

Resource usage: If careless, the team might create a large volume of tests at a high frequency, which might generate enough load to degrade the application. It is up to the team to properly balance the volume of tests and their frequency.

A.8.8 *Related Patterns.* EXTERNAL MONITORING providers expose APIs which can be used to programatically manage the tests. A team that adopts INFRASTRUCTURE AS CODE will be more efficient at managing their tests.

EXTERNAL MONITORING can be used to feed information for LOG AGGREGATION, facilitating a centralized view of the issues observed in the application from this monitoring strategy as well.

HEALTH ENDPOINT MONITORING from Microsoft is similar to this pattern proposes the creation of HTTP health checks exposed by the application, so that an external tool can verify the application status [Microsoft 2017a]. That implementation differs from EXTERNAL MONITORING, as it requires specific endpoints to be implemented and tested from the external health checking tool. Instead, EXTERNAL MONITORING proposes that the external tool interacts with the application as a client would, using any public interface, not limited to HTTP, validating that it is providing the expected answers.

The COLLABORATIVE MONITORING AND LOGGING pattern [Erl et al. 2015] describes how monitoring and logging activities can be coordinated between a cloud consumer and provider, describing that monitoring and auditing requirements can described by the consumer but observed by the provider. This approach is similar to

External Monitoring, given that the monitoring behavior is extracted from the application the consumer is developing and executed with an external tool, managed by the cloud provider.

A.8.9 *Known Uses.* Multiple services are available providing the External Monitoring tool required to implement this pattern. StatusCake, Pingdom or NewRelic [Relic 2017; Pingdom 2017; Statuscake 2017] are only three of those applications. Pricing and features set them apart, with most being able to test at the HTTP and TCP layers.

A.8.10 *Further Considerations.* Juvenal, a first century poet, in his *Satires* series of books wrote the famous Latin quote "Quis custodiet ipsos custodes?", roughly translated to *who watches the watchmen?* [Winstedt 1899]. This quote can still today motivate discussion around cloud monitoring. By relying on an external tool to monitor the system, we are delegating the responsibility of capturing failures to an external system. What must be taken into consideration is that the external system is a piece of software as well, which might also. In such scenario, a failing system would not be detected, given that the monitoring system would also be unavailable.

A.9 Messaging System

As service instances increase, communication between services needs to be abstracted, enabling proper balancing between instances. This communication strategy is required to be fault-tolerant and scalable to maintain the application's resiliency. As a solution, a messaging system, colloquially known as message queue, can abstract service placement and orchestrate messages with multiple routing strategies between them.

A.9.1 *Context.* The adoption of microservices as an architectural style introduced the need for services to cooperate in a decentralized and possibly unreliable environment. It is not guaranteed that every component is online at all time, nor that each service has a stable IP address (Internet Protocol) or number of instances running.

These intricacies of cloud computing introduce several requirements, namely, services need to *communicate* with each other in an ever-changing environment, the communication process must be fault-tolerant, ensuring that the system as a whole is *resilient* when confronted with irregular behavior from either side of the communication, and message passing should be *asynchronous*, *decoupled*, *evolvable*, using a *content-agnostic* communication channel.

A.9.2 *Example.* Consider an home automation solution that manages Air Conditioning (AC) systems. Three services compose the solution: *Sensor Reader*, *Data Receiver* and *AC Manager*. *Sensor Reader* is deployed inside the user's house. It is responsible for acquiring and forwarding temperature data. *Data Receiver* is a Web Server that receives temperature metrics and persists them in a database. *Data Receiver* is also able to provide aggregations over the data persisted in the database. *AC Manager* is responsible for managing AC units by evaluating the average temperature over the course of the past 10 minutes, configuring an AC to generate cold or warm air. The three services must cooperate to provide a complete solution for automated AC management. The expected interaction between them is depicted in figure 17.

A.9.3 *Problem. As service instances increase, communication between services needs to be abstracted, enabling proper balancing between instances. This communication strategy is required to be fault-tolerant and scalable to maintain the application's resiliency.*

Services in a cloud application need to communicate with each other to cooperate. A common communication strategy uses a client-server approach, limiting the communication to the two intervening service instances and requiring that the client knows how to connect to the server, namely its hostname and server port. Cloud application

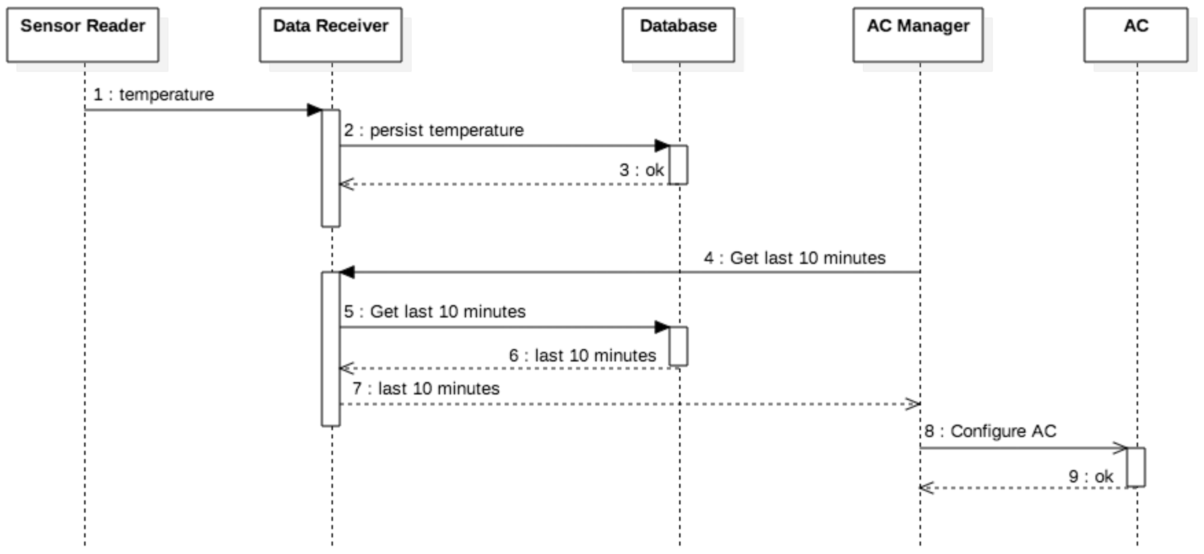


Fig. 17: A microservice architecture-based system to capture and persist temperature metrics from an home environment, later used to configure an AC system. The arrows in the sequence diagram represent the massages exchanged between the components.

are deployed into dynamic hardware, which means that internal server's addresses are not available during development time, rendering troublesome to use direct communication between services. Furthermore, when multiple instances of a service exist, the traffic needs to be balanced between all instances.

Considering the above, the need for an abstraction over the communication between services is identified. Such channel must enable passing any type of messages and correctly identify the sender and receiver of such messages. Such communication channel must be scalable, ensuring that latency requirements are met even when handling large volumes of messages.

A.9.4 *Forces.* The following forces, represented in Figure 2, need to be balanced while considering the adoption of this pattern:

This pattern is influenced by the following forces:

Decoupling: A sender doesn't need to know the network address of a receiving service to communicate with it.

Scalability: The communication channel needs to be itself scalable.

Resilience: Communication should be resilient, despite failures in the communication channel.

Persistency: Messages between services should be persisted until there is a confirmation that they have been processed.

Structure Agnostic: The communication channel should be agnostic to the messages it orchestrates.

Dynamic and Flexible: The topology of the system will evolve with time, with new services joining existing ones, and others leaving in real time.

Payload security: The communication channel should support encrypted messages.

Channel security: The communication channel should itself encrypted.

Latency: Introducing an indirection in communication increases the latency required for passing a message between two services.

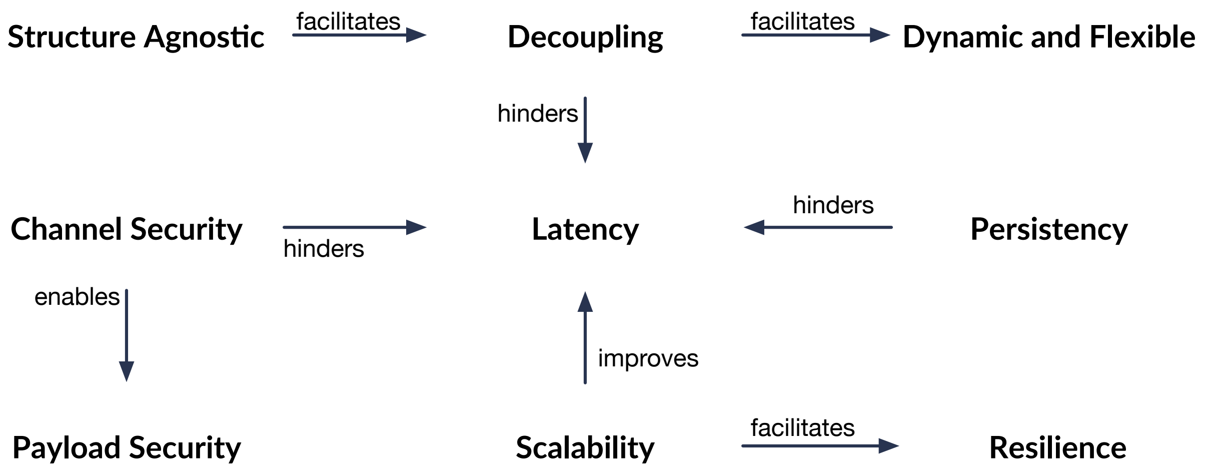


Fig. 18: Relationship between the MESSAGING SYSTEM forces.

A.9.5 *Solution. Use a* MESSAGING SYSTEM*, colloquially known as message queue, to abstract service placement and orchestrate messages with the optimal routing strategy between them.*

A MESSAGING SYSTEM is responsible for routing messages between services which can be both producers and consumers of messages. Messages can vary in size and contents, given that the channel is agnostic of their internal structure, as long as they respect the adopted protocol.

MESSAGING SYSTEM works by creating one or more queues that work as a first in, first out (FIFO) data structure. Some implementations provide the possibility of prioritizing messages in the queue. Quality of Service (QoS) policies can also be applied, forcing consumers to confirm that they have successful processed the message before it gets discarded from the queue. QoS ensures that a failing service won't remove a message from the queue without it actually being processed. If a service fails to acknowledge that the message has been processed in an acceptable time period, the message becomes available for another consumer to process.

Most implementations support multiple message delivery strategies. RabbitMQ, which is one of the most adopted implementation, supports simple queues, exchanges with multiples queues, routing, topic-based consumption and RPC [Pivotal 2007].

When implementing Remote Procedure Call (RPC), services can issue requests to the message queue and block waiting for an answer. A consumer would pick up the request, process it, and send it back to the queue, destined to the request sender. That first server would then receive his request and resume his computation.

Moving the responsibility of handling all communications to the Message Queue service makes it a single point of failure. For this reason, messaging services are typically deployed with redundancy, ensuring that communications between services will continue to work if some instances fail.

The concept of message passing systems has been available for several years, as middlewares that provide highly-observable communication strategies, namely one-to-many communication, providing dynamic connections among services. Initial reference to messaging applications as a mean of communication between servers was first introduced on the 2001 patent *Message Queue Server System* [Yarbrough and Hook 2002]. More recently, several standards have been introduced, namely the *Advanced Message Queuing Protocol* (AMQP) and the *Message Queue Telemetry Transport* (MQTT) [Magnoni 2015].

Most implementations will enable the communication channel to use an encryption algorithm to protect the communication channel. Being agnostic to the message's contents, the payload itself can also be encrypted when needed, preventing data leaks even if the MESSAGING SYSTEM is compromised.

A.9.6 *Example Resolved.* Considering the example described in section A.9.3, the three services can communicate using a message queue based distribution in a message system, as shown in figure 19. Message queues can be identified by a name and require consumers to subscribe the queues from which they want to receive messages.

Initially, the *Data Receiver* service would subscribe to queues *metrics* and *requests*. *AC Manager* would subscribe to a queue named after it, *manager*.

Inside the house *Sensor Reader* would capture temperature metrics and send them to the message queue using the *metrics* queue. Asynchronously, *Data Receiver* would consume these messages and persist them in the database.

Periodically, *AC Manager* would require the last 10 minutes of temperature metrics to the message queue in the *requests* queue. *Data Receiver* would consume that message, gather that information from the database and sent back to the message queue using the *manager* queue. Finally, *AC Manager* would consume those messages and configure the AC system with the appropriate behavior.

A.9.7 *Resulting Context.* In the context of engineering software for the cloud, message queues can abstract where services are located, eliminating the need for discovery mechanisms between them. Each service can communicate directly with one or more queues, requiring only the address of the Message Queue service.

Using message queues also facilitates service scaling. Services receiving traffic from outside channels should be scaled in order to handle the traffic. These would then inject messages in queues which are being consumed by other services. In such architecture, the size of the queue can be used to understand if and how a service should be scaled, aiming at always keeping the message queue as small as possible.

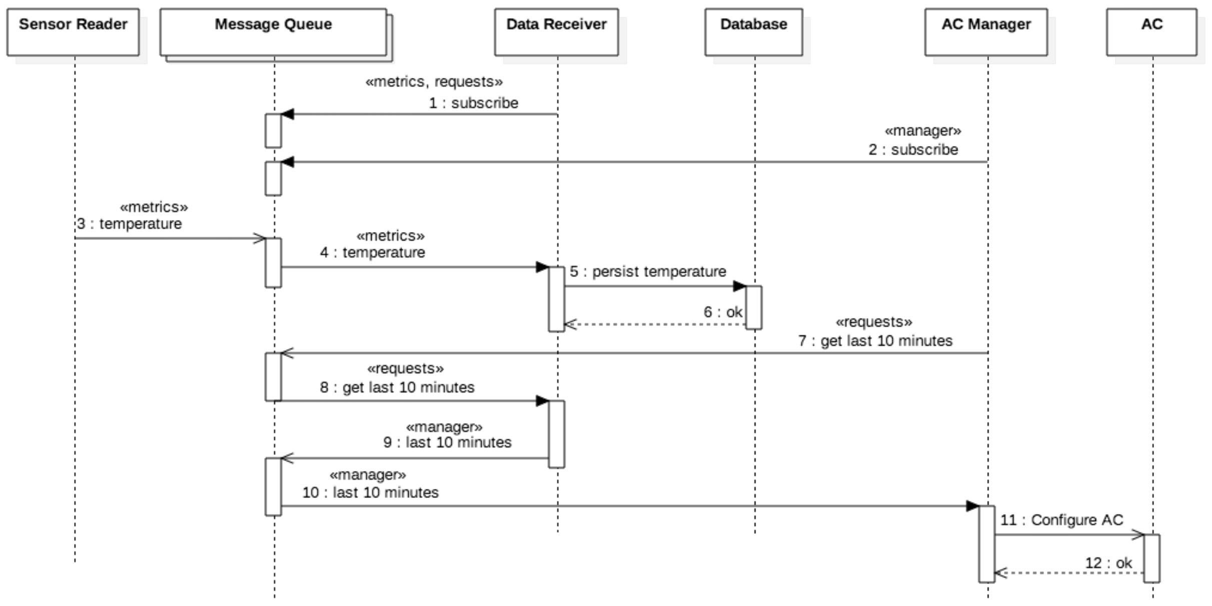


Fig. 19: Communication between the three described services, routed via a messaging system. No two services communicate directly. Arrows represent the messages exchanged in the systems.

This pattern can positively improve a cloud application as follows:

Decoupling: services allow a faster integration of new services in the ecosystem.

Scalability: is achieved by creating an infrastructure of Message Queue services, proportional to the number of services using it.

Resilience: is improved as messages can be kept in the queue until a consumer service is available to process them. The message queue software might also be deployed in an infrastructure and keep the messages persisted in disk to improve its own resilience.

Availability: for messages is ensured, with the message queue being able to persist the messages as long as needed until these are consumed.

Security: is improved by obscurity, as the services receiving messages do not need to be reachable from the message sending services. Also, the communication channel use encryption to enforce a secure communication of all messages sent through it.

On the other hand, the following pitfalls are observable:

Complexity: , by increasing the level of indirection, understanding how messages are being passed between service might become incredibly complex and hard to debug. For this reason, Facebook's Flux architecture which is partially event driven, explicitly disallows sending nested events.

Latency: is increased, since an additional hop is required to get a message from its producer to the consumer. Modern message queues, when co-located with both services and given the appropriate network conditions, can still ensure latency under 50 milliseconds.

Single point of failure: Without a redundant deployment, a failure in the messaging system will halt all interaction between the services.

A.9.8 *Related Patterns.* Message Queues are a more elaborate approach to Hohpe's Message Buses, which provided a basic communication channel between applications. In his book *Enterprise Integration Patterns*, additional communication patterns that most message queue implementations have adopted are described, such as PUBLISH-SUBSCRIBE CHANNEL or GUARANTEED DELIVERY [Hohpe and Woolf 2003].

Another version of the Publisher-Subscriber pattern was also documented by [Bushmann et al. 1996].

This pattern introduces an approach to allow services to communicate without knowing their peers location. This might not be acceptable at all times, mostly due to latency constraints. For those cases, LOCAL REVERSE PROXY [Boldt Sousa et al. 2015] can be applied.

A similar strategy described by the IO GATEKEEPER and related patterns in the telecommunication domain for managing the interaction between humans and systems [Hanmer 1998].

LOCAL REVERSE PROXY can also be used to discover where the message queue is available in the infrastructure.

Messaging systems can be used to implement LOG AGGREGATION, by having services communicating their logs as messages, which are then aggregated by the log centralization service.

A.9.9 *Known Uses.* MESSAGING SYSTEM has a wide range of adoptions. At CERN, it was used to make information available for multiple monitoring tools in multiple projects, namely in the LHC [Casey et al. 2011]. A similar environment to the one presented in section A.9.6 is described by Grgićm, along with details on how to instantiate it [Špeh and Heđ 2016].

In another example, [Herrmann et al. 2016] demonstrates how message queues can be adopted to acquire real-time data from trains and be used with Reactive Blocks[6] to facilitate collaboration in development and maintenance of software systems.

[6]Project details available at www.bitreactive.com.

A.10 Local Reverse Proxy

Services might lack the network information required to communicate with other dynamically-allocated services. Communication can be achieved by abstract service network details by defining a service port for each service. Use a reverse proxy to expose that port in all servers, forwarding traffic to where the services are deployed.

A.10.1 *Context.* Cloud applications are commonly composed by a multitude of services, which may be spread over multiple physical servers in different networks. In order for services to cooperate they need to know how to contact each other, which implies the need for configuration or discovery of the hostname or IP and port where the required service can be reached. Furthermore, when a service has multiple instances, required in high availability setups, there might be the need to evenly distribute traffic between existing instances.

A.10.2 *Example.* An application server receives HTTP requests and queries a database server to get the required information to process the response. For scalability purposes, the database is distributed and the number of instances varies according to the average system load. The application server will have to query the database but, as it is running on an dynamically provisioned hardware, the application has no information about how the database servers can be reached. Figure 20 represent a possible distribution of services among the existing servers of such system.

A.10.3 *Problem. Services might lack the network information required to communicate with other dynamically-allocated services.*

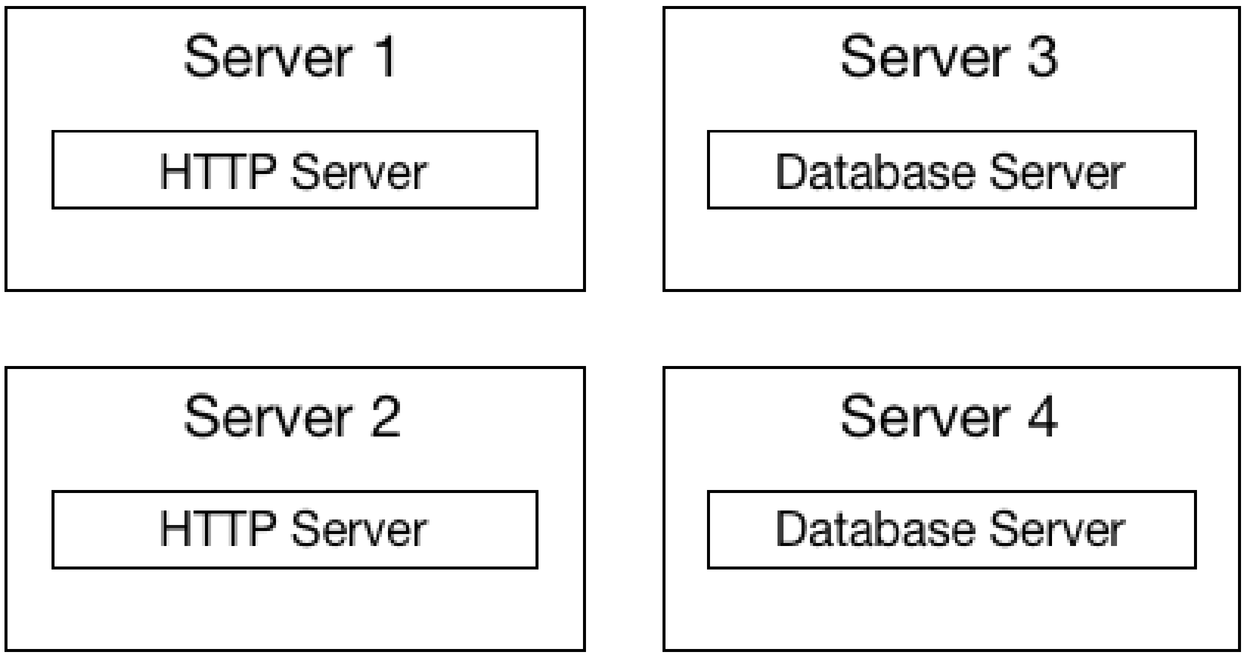


Fig. 20: The four members of an infrastructure, each hosting a service.

Service decoupling is required as software gets deployed and scaled automatically in the cloud, enabling the scaling of individual software components using dynamically provisioned hardware. Deploying in these conditions leave the services unaware of where their dependencies are allocated, requiring a discovery method to enable communication between them.

A.10.4 *Forces.* The following forces, represented in Figure 21, need to be balanced while considering the adoption of this pattern:

Real-time discovery: State mus be updated when there is a change in the number of instances in a service.

Location decoupling: Services do not need to know where others are deployed to communicate with them.

Protocol Agnostic: Work at the network level, supporting any protocol adopted by the services.

A.10.5 *Solution. Abstract service network details by defining a service port for each service. Use a reverse proxy to expose that port in all servers, forwarding traffic to where the services are deployed.*

For each service, a service port is set, which is a port that will always be available in all servers, exposed by a reverse proxy tar will forward traffic to one of the available services. Network proxies work at the network level, which makes them protocol agnostic, forwarding TPC, UDP, HTTP or any other traffic type.

The reverse proxies need to be updated every time there is a change in the services in the infrastructure. There are multiple strategies for doing so. One is to have a service registry where each service announces itself, along with a dedicated software that periodically reads this information and updates the proxies. Another alternative is to query this information from an ORCHESTRATION MANAGER.

When multiple instances of a service are available, it is up to the local proxy to decide how to distribute these requests, acting as a load balancer. The balancing algorithm might be configurable, for example, distributing the requests using a round-robin technique or in a smarter way, according to the target's resource availability.

Whenever a service goes down, the local proxy will forward the request to another instance of the same service or refuse the connection if no instance is available. In the later scenario, the source service needs to use a retry mechanism to keep trying the service until it becomes available.

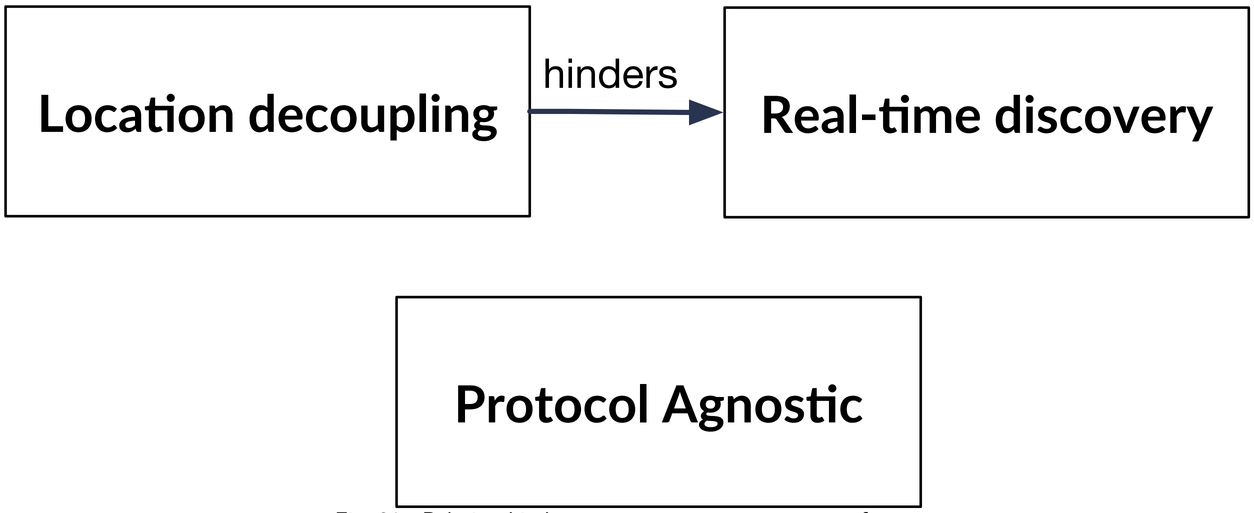


Fig. 21: Relationship between LOCAL REVERSE PROXY forces.

A.10.6 *Example Resolved.* This technique requires an external orchestration mechanism to keep meta-information on the services running in the infrastructure, regarding hosts and ports. Each host machine has a local proxy that periodically queries the orchestration manager and forwards a known local port to the host(s) and port of where a service available in the infrastructure. The applications expect a specific port to be available locally that will abstract the exact port and host where the service is actually running. Consider the example previously described: a web application is deployed with two HTTP Servers receiving external requests, which must communicate with one of the two other Database Servers to create a reply. For the HTTP servers to communicate with the database, instead of establishing a direct connection, they connect to the local known port, leaving for the proxy to forward the request to an available Database server. Scalability is achieved by varying the number of Database or HTTP Servers independently, relying on the Local Proxies on the HTTP side to properly identify available Database Servers and distribute load between them. This example is represented in Figure 22.

A.10.7 *Resulting Context.* This pattern introduces the following benefits:

Real-time discovery: Changes to the infrastructure are immediately identified by the orchestration manager, which will reconfigure the local proxies.

Location decoupling: Service development can ignore the actual physical location of other services it is integrating with, relying on the local reverse proxy to forward traffic to where the service is executing.

Protocol agnostic: Proxies work at the transport OSI layer or lower, hence, are protocol agnostic.

The pattern also introduces the following liabilities:

Monitoring: A mapping between a service and its running instances must be maintained at all time so that the reverse proxies are properly configured and only redirect traffic to active services.

A.10.8 *Related Patterns.* This pattern may be applied when CONTAINERIZATION is being used to isolate applications, facilitating communication between containers hosted in different servers, without requiring applications to individually integrate with discovery mechanisms. Information about service ports in each container can be injected using environment variables.

This pattern depends on an external mechanism that keeps track of each service in the infrastructure. An ORCHESTRATION MANAGER holds this information and could be queried for it.

A.10.9 *Known Uses.* A basic approach is presented by Wilder, keeping an Nginx reverse proxy updated according to meta-information extracted from running docker containers in the local machine [Wilder 2015].

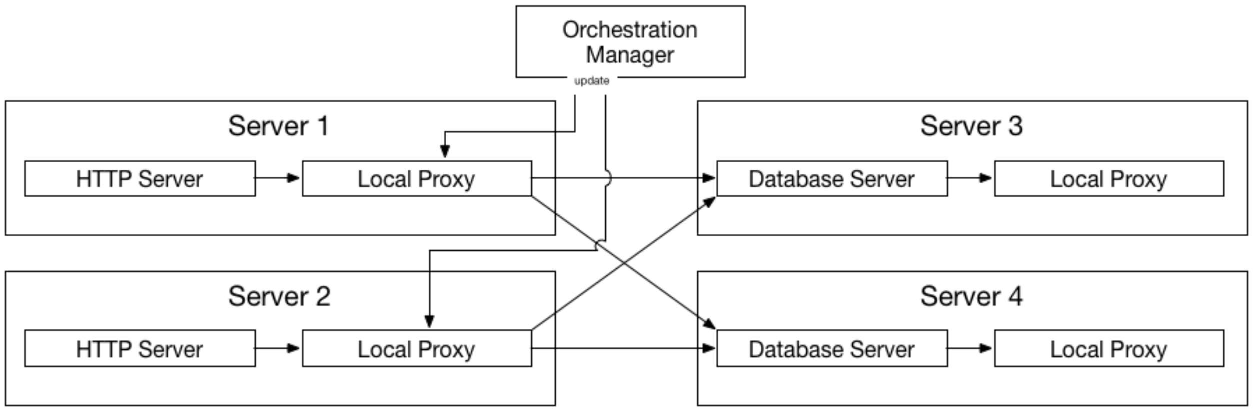


Fig. 22: Local Proxy configuration example.

The reverse proxy Vulcanproxy [Community 2015b], together with the distributed key-value storage Etcd [Community 2015a] provides a reverse proxy service agnostic to the software using it. By depending on Etcd, it is not an optimal solution as it requires services to register themselves with Etcd.

A better solution is based on Apache Mesos [Foundation 2015] which allow jobs to be spawned across multiple nodes, managing their allocation and Marathon, an infrastructure-wide init and control system for Mesos [Mesosphere 2015a]. Using meta-information available with Marathon, a script can periodically update a local proxy server on each machine in the infrastructure, forwarding a TCP or UDP port, named the service port, to the actual address where the application is running, despite it being local or in a remote machine [Wuggazer 2015]. There are many implementations available to work with Marathon, including Bambo, an HAProxy auto-discovery and configuration tool for Marathon [Qubit 2015]. There is also a script that can configure a local HA proxy, made available by Marathon's team [Mesosphere 2015b].

pending PLoP 18 details