Engineering Software for the Cloud - Patterns and Sequences

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

Software businesses are quickly moving towards the cloud. While cloud computing is not a new research topic, engineering software for the cloud is still a challenge, requiring broad knowledge over a multitude of processes and tools that most software development teams lack. This paper intends to identify relevant practices required to efficiently engineer software for the cloud. The authors use patterns to capture and share those practices and describe their possible usage in exemplar sequences. Patterns are aggregated into several categories: development, deployment, execution, discovery and communication, monitoring and supervision. The targeted audience is identified, as well as three sequences of applications for the patterns. The paper is targeted at newcomers, practitioners and experts at cloud development for guiding through all architectural decisions, solving specific issues or just validating previous decisions respectively.

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

General Terms: Design

Additional Key Words and Phrases: Cloud Computing, DevOps

1. INTRODUCTION

Software is being built at a global scale, with over 46% of the world population having access to the Internet [Internetlivestats.com 2016]. Cloud Computing is facilitating this revolution, providing computation as a commodity that can be acquired on a pay-as-needed basis, just like water or electricity [Mirashe and Kalyankar 2010].

This versatility enabled individuals and small teams to become competitive against large corporations without requiring prohibitive investment in infrastructure, software development and human resources, democratizing access to the market and truly enabling anyone to develop global Internet-based businesses.

The Cloud is today the preferential channel for businesses, enabling smaller players to be competitive at a global scale by deprecating the infrastructure investment requirement.

Despite widely available, Cloud Computing is not trivially adopted. Exploiting the advantages of Cloud Computing requires a complex paradigm shift from traditional software development in distinct areas, such as software architecture or team organization. With the possibility for individuals and small teams to create highly scalable software through Cloud Computing, writing software is no longer the single required component for successfully creating a software business. Automating quality assurance and operations is equally as important [Sousa et al. 2015; Roche 2013].

The research required to fully describe all the patterns that are relevant to develop software for the Cloud is extensive, and this work is just the first step towards a complex pattern language that can help researchers and practitioner. This initial work identifies the most relevant patterns and provides an high-level description to them.

This work leverages a pattern language as a framework that captures the required knowledge to bootstrap and maintain software for the Cloud. The usage of patterns, introduced by Christopher Alexander, has been widely accepted as a way to capture and share practical knowledge and experience.

This paper is structured in four sections. The second section to this paper describes the identified patterns. Section three demonstrates the application of the patterns as a sequence of decisions and actions, enabling the reader to better understand their relation. The last section presents the conclusions and future work.

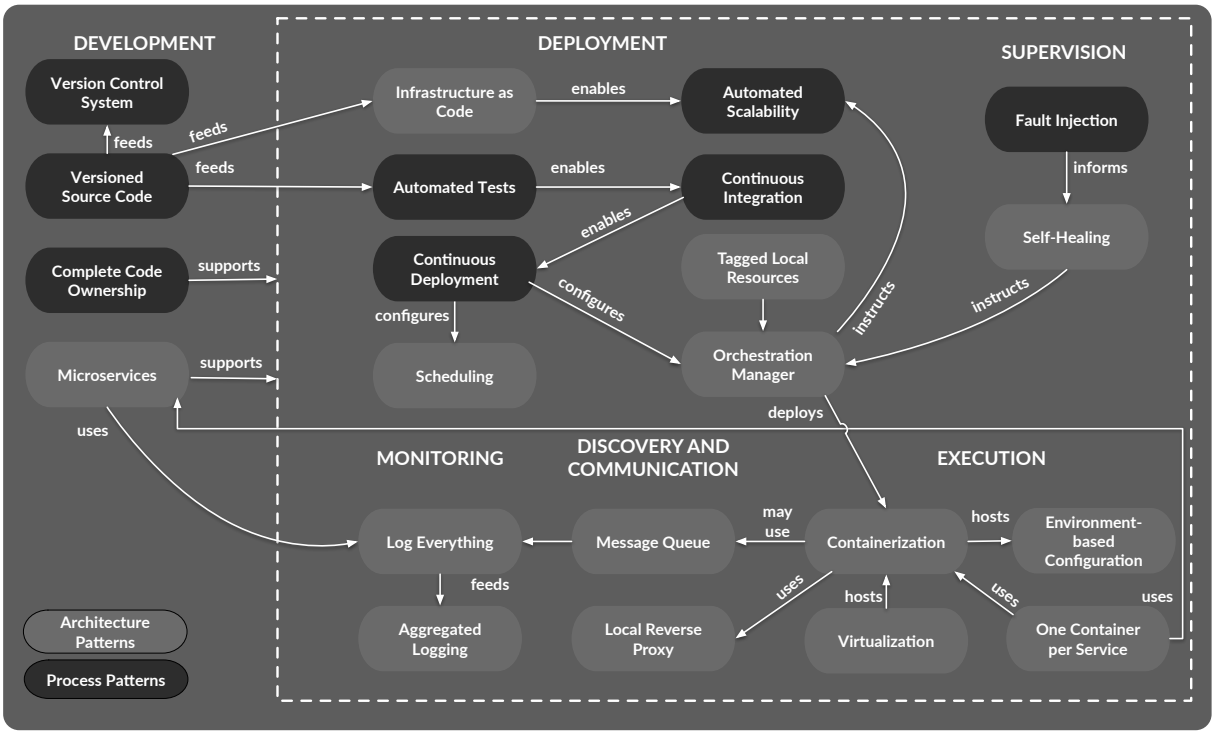

Fig. 1. Pattern map, its categories and patterns, with most relevant relations.

2. PATTERNS FOR ENGINEERING SOFTWARE FOR THE CLOUD

Developing software for the Cloud is not trivially achieved due to the complex paradigm shift it requires from traditional software development. The application itself is no longer the only software being implemented, development teams also have to ensure that the whole ecosystem is automated programmatically in order to achieve success in a time and cost efficient way. This is mostly motivated by the need for modern software to scale. By migrating to the Cloud, software becomes immediately exposed in the World Wide Web, possibly achieving global visibility. Programmatically automate operations is essential for creating highly scalable, resilient applications in a cost-efficient way.

New practices, tools and services are being made available every day for engineers to develop their software for the cloud. For an inexperienced team, multiple man-month of research might be required to achieve a functioning setup. Several man-years might be needed for creating a fully automated Cloud orchestration setup.

This section presents patterns that aggregate the required knowledge to develop software for the cloud. The patterns are aggregated into six categories: DEVELOPMENT, EXECUTION, INFRASTRUCTURE, MONITORING, SUPERVISION and DISCOVERY AND COMMUNICATION, as shown in Figure 1. These patterns can be used separately to validate or facilitate the acquisition of knowledge on a specific practice, or together, guiding non-experts through the process of building software for the cloud efficiently.

2.1 Development Patterns

These patterns are observable during the development phase and are mostly relevant for the development team. Following them will enable the team to become more efficient at writing their source code, as well as reduce the time spent at manual tasks.

Microservice Architecture. It is difficult to scale the development and execution of large monolithic applications. Such application can be divided in many small services, called Microservices, that are developed and deployed independently. Each Microservice cooperates with other Microservices to provide complex Cloud Applications [Lewis and Fowler 2014; Namiot and Sneps-Sneppe 2014; Richardson 2014].

Versioned Source Code. Millions of lines of code are produced by multiple developers while implementing large applications. The development process is prone to errors that might result in the loss of a valuable part of the code when changes in the source code history are not kept. Having the source code versioned allows developers to understand who made which changes and quickly jump back to any code revision.

Version Control Systems. Software teams can range from one to hundreds of developers, possibly geographically spread. While developing software collaboratively, there is the need for developers to share and integrate their code. A Version Control System (VCS) provides the required platform for doing so. It allow a developer to browse or download the entire code base history, as well as uploading his own changes to it. Additionally, most VCS provide support for code merging, issue tracking and more, which greatly simplify the team collaboration.

Automated Tests. While developing, it is common for new bugs, and sometimes others which were previously corrected, to be introduced in the code. Automated testing should be adopted as part of the development process, having tests being written as part of the development process in such way that enables their automated execution and ensures that new software is behaving as expected and that no previous bugs are observed again.

Continuous Integration. Code fragmentation might happen when many developers are working on the same code base, rendering troublesome the process of fixing conflicts and integrate separate developments. To prevent such issue, code changes should be granular and immediately integrated with the remaining code. To be accepted, changes to the code should be tested for regressions using automated tests. Additionally, peer-review might be adopted to guarantee that multiple team members are aware of the change and that the implementation meets the expected quality.

Infrastructure as Code. Manual infrastructure management requires human intervention on every change, rendering it hard to trace changes and evolve the infrastructure. Implementing operations as code should be adopted as part of the development process, making all changes to infrastructure programmatic.

Continuous Deployment. . Often, new implementations take considerable time from being ready to reach the production environment. This affecting time to market and is mostly motivated by the need for manual intervention. New features should be deployed as soon as they are ready. Right after the Continuous Integration phase, changes to the code have been validated and can be automatically deployed.

Complete Code Ownership. . Some companies have multiple teams with different roles working on the same project, preventing a single team o take full ownership of the product. Keeping these teams efficient is not trivial, as they might not share the same agenda at all times. A single team should have full ownership over a project, being responsible not only for the development, but for the entire life cycle of their software and its orchestration. Quality and operations should be fully automated and implemented as part of the development process.

2.2 Execution Patterns

These patterns describe the process of packaging and getting services deployed.

Virtualization. Software requires an host environment where it will be executed. In the past, configuring this environment required the acquisition of hardware, setting up the operative system, installing all dependencies and then the software itself. Today, Platform as a Service (PaaS) is available from most Cloud Providers, enabling the creation and deletion of virtual machines on demand using an API, bootstrapping as many virtual machines as required with all the required resources to run the software. Private solutions also exist doing the same on top of a private bare metal cluster, using a similar API.

Containerization. Creating development and production environments manually is a difficult process to scale. Even by adopting Virtualization, virtualizing an entire computer very resource-demanding. Also, creating execution environments for Software requires the installation of all its dependencies, polluting the host. Software containers can be used to create immutable, reproducible, portable and secure environments for executing software. Containers prevent polluting the host environment with dependencies and configurations and are easily scalable, as there is no need to virtualize the Operative System layer in each one of them, providing little virtualization overhead [Sousa et al. 2015; Scheepers 2014].

One Container per Service. Services should be isolated from each-other, preventing secondary effects in their behaviors. Leveraging a Microservice based architecture, each service should be deployed in its own container, providing complete software isolation and scalability.

Environment-based Configuration. While developing Microservices, they should facilitate the service's portability. Services inside containers should acquire their configuration from environment variables that are made available to the container in for each specific execution environments. This enables not only the service's configuration to be injected into the environment, but the host resourcesâĂŹ details to be made available in the same way.

2.3 Infrastructure Management Patterns

Infrastructure empowering Software in the Cloud is typically volatile and dynamically allocated. These patterns describe how to setup the necessary hardware and software to get services online.

Tagged Local Resources. Different servers might have different hardware, and some might be more appropriate for the allocation of services than others. Servers should make their resource information available for an orchestration manager, enabling it to make informed decisions regarding service allocation.

Orchestration Manager. Services differ in requirements. 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. Again, the allocation of services must be carefully evaluated to ensure proper behavior. An orchestration manager can be responsible for allocating services to the proper hosts, considering their overall and available resources, leveraging the allocation of other services.

Scheduling. Scheduled operations are often required for performing assynchronous tasks, such as database maintenance, sending emails or performing backups. These might run at a given frequency or on a single point in time. A scheduler should orchestrate the execution of these programs in a cluster and evaluate their outcome, generating error reports when need.

Automated Scalability. Cloud applications can quickly move from being almost idle to serve millions of users, requiring the infrastructure to scale. In a cluster, resources should be monitored and used to decide when to scale the system automatically. In a Microservices Architecture, each component should be scaled individually, leveraging costs and service quality.

2.4 Monitoring Patterns

Monitorization continuously evaluates the conditions at which the services are being executed. Monitoring can work reactively, detecting issues on data generated by the application, such as a log entry or an alarm, as well active by interacts with the services directly, evaluating their correct behavior.

Log Everything. Debugging an application requires as much information as possible in order to trace the actions that lead to an issue. Services should produce verbose execution logs that should be kept for the longest period of time possible.

Aggregated Logging. Developer have to leverage multiple log files, possible from multiple sources, in order to trace an issue. To prevent this, the team should have a centralized view over the logs generated by all services from every host, allowing them to query and aggregate them as needed in a single point.

2.5 Supervision Patterns

Supervision ensures that services run as expected, executing the proper action to recover them in case of failure.

Self-Healing. Services running inside containers should be resilient. In case of failure, exploiting the immutability of containers, the service shall restart automatically to try to recover the service. Additional methods might be introduced, with the Orchestration Manager possibly deciding on the best recovery strategy.

Fault Injection. Software issues in production often appear in unexpected scenarios. To ensure the system's reliability and resiliency, a fault injection mechanism can periodically or continuously inject random issues in the system, evaluating if it continuous to behave appropriately. Fault injection can evaluate reliability by injecting unexpected values into the service and observing if any unexpected behavior occurs. Resiliency can be tested by randomly shutting servers down, ensuring they scale right back up without impacting service quality.

2.6 Discovery and Communication Patterns

Supports services to discover each other, simplifying Master-Slave elections or message-driven work distribution.

Local Reverse Proxy. Microservices in a cluster need to communicate between themselves. While using an Orchestration Manager, container allocation is dynamic, preventing service location to be made available during compilation. The exact location can be abstracted through a service port exposed locally on every machine that is properly forwarded to one instance of the service, possibly balancing traffic between multiple instances [Schumacher et al. 2006].

Message Queue. Services must be able to communicate amongst themselves both synchronously for RPC and asynchronously for delegating information to collaborating services. A message queue can be used to send both types of messages between micro-services, eliminating the complexity associated with service discovery [Gawlick 2002].

3. ADOPTING THE PATTERN LANGUAGE

Resistance to change is by itself a pattern. Adopting a pattern language for developing software for the Cloud introduces the need for teams to change their mindset regarding their processes and software architecture. This section briefly identifies the target audience and describe how their experience would influence the way they would use the pattern language. Finally, this section presents a set of sequences that exemplify how a given persona with a specific challenge and level of experience would use the pattern language.

3.1 Target Audience

The authors identified three different target audiences as possible adopters for the pattern language. Each audience would use the pattern language differently and can be described as follows:

The Newcomer. A person or team that wants to develop software for the Cloud without having any previous experience. The Newcomer would benefit the most out of the pattern language, possibly influencing every decision he makes for his software architecture.

The Practitioner. A person or team that has some experience at developing software for the Cloud. He would use the pattern language to overcome a specific problem, as a reference for addressing new challenges, as well as to validate his other decisions.

The Expert. A person or team that is generically experienced at with Cloud Computing, or an expert in a particular sub-field. The Expert could use the pattern language as a validation tool for his own knowledge or as a tool to broaden his knowledge.

3.2 Sequences

Sequences help to understand how the patterns relate and complement each other. This section was inspired by *The Unfolding of a Japanese Tea Garden* by Christopher Alexander [Alexander 2006] and leverage the pattern language to solve three possible problems common while developing software for the Cloud.

3.2.1 *Containerizing the Services.* A Cloud *Practitioner* is having trouble managing the Virtual Machines that host his services, due to Software versions incompatibilities.

The *Practitioner* could adopt VIRTUALIZATION to provide hosting virtual machines and CONTAINERIZATION and ONE CONTAINER PER SERVICE to deploy his services in an isolated and scalable way. This would enable him to decide what resources he needs to execute his services, deploy the right number of virtual machines and then host the containers in them. Using an ORCHESTRATION MANAGER within his virtual machines cluster would allow him to deploy the containers in the hosts, guaranteeing that the required resources are available in the host for running the container.

3.2.2 *Microservice Scalable Cloud Application.* A *Newcomer* team wants to develop a scalable Web Application. They require the application to scale automatically based on the servers load.

Considering the pattern language, this team should start by developing the Web Application's components using a MICROSERVICE ARCHITECTURE. Their scaling operations should then be implemented using INFRASTRUCTURE AS CODE, facilitating the manipulation of the infrastructure programatically. The ORCHESTRATION MANAGER would then continuously evaluate the cluster's status, instructing the AUTOMATED SCALABILITY component to scale the cluster up or down.

3.2.3 *Backing up the Database.* A *Practitioner* has just adopted CONTAINERIZATION and an ORCHESTRATION MANAGER. He now needs to periodically backup his databases to a remote storage.

The *Practitioner* could implement the backup logic in a script inside a container. By adopting the SCHEDULE pattern, that container could periodically be started within the cluster by the ORCHESTRATION MANAGER.

4. SUMMARY AND FUTURE WORK

This paper introduces the structure for building a pattern language for building Software for the Cloud, aiming at helping software development teams to build their Cloud Software. The authors presented a pattern language composed by 21 patterns and three sequences that demonstrate how a team could leverage the patterns in different scenarios. Future work will continue the development of the pattern language by thoroughly detailing each pattern, as well as complementing it with additional patterns. The authors than plan to validate the language in an experiment with development teams from the industry.

REFERENCES

ALEXANDER, C. 2006. *The Nature of Order: The Process of Creating Life.* Center for Environmental Structure.

GAWLICK, D. 2002. Message Queuing for Business Integration. *{eAI} Journal* October 2002, 30–33.

INTERNETLIVESTATS.COM. 2016. Number of Internet users in the world.

LEWIS, J. AND FOWLER, M. 2014. Microservices.

MIRASHE, S. P. AND KALYANKAR, N. V. 2010. Cloud Computing. *2,* 3, 78–82.

NAMIOT, D. AND SNEPS-SNEPPE, M. 2014. On Micro-services Architecture. *International Journal of Open Information Technologies 2,* 9, 24–27.

RICHARDSON, C. 2014. Pattern: Microservices Architecture.

ROCHE, J. 2013. Adopting DevOps Practices in Quality Assurance. *Communications of the ACM 56,* 11, 38–44.

SCHEEPERS, M. 2014. Virtualization and Containerization of Application Infrastructure: A Comparison.

SCHUMACHER, M., FERNANDEZ-BUGLIONI, E., HYBERTSON, D., BUSCHMANN, F., AND SOMMERLAD, P. 2006. *Security Patterns: Integrating Security and Systems Engineering*.

SOUSA, T. B., CORREIA, F. F., AND SERENO FERREIRA, H. 2015. Patterns for Software Orchestration on the Cloud. In *Proceedings of the 2015 Conference on Pattern Languages of Programs*.