UNIVERSITY OF TURKU

# Using Infrastructure as Code for Web Application Disaster Recovery


Communication Systems

Master's Degree Programme in Information and Communication Technology

Department of Computing, Faculty of Technology

Master of Science in Technology Thesis

Author:

Tommi Tuomisto

Supervisor:

Jussi Haaja (Ambientia Group Oy)

Examiners:

Seppo Virtanen (University of Turku)

Tahir Mohammad (University of Turku)



June 2022

**Master of Science in Technology Thesis**
**Department of Computing, Faculty of Technology**
**University of Turku**

**Subject**: Communication Systems
**Programme:** Master's Degree Programme in Information and Communication Technology
**Author**: Tommi Tuomisto
**Title**: Using Infrastructure-as-Code for Web Application Disaster Recovery
**Number of pages**: 58 pages, 4 appendix pages
**Date**: June 2022

Legacy, industry established disaster recovery approaches are known for impeding a relatively high additional expenditure, thus limiting the usage of such mechanisms only to the most business-critical IT systems and applications. However, with the emergence of Infrastructure-as-Code practices, this paradigm can now be challenged.

The objective of this thesis is to design and implement a novel disaster recovery tool, that can be used for the recovery of a web application. By following the design science methodology, this thesis proposes a primary-fallback oriented disaster recovery model, where the fallback site of the infrastructure is an empty cloud service account, into which a near duplicate copy of the primary site is recreated in the event of a disaster.

The proposed recovery process consists of two phases, where the 2nd phase stateful application data recovery procedure is kept as an add-on functionality to the 1st phase stateless infrastructure management practices. For switching from primary to fallback site, the design proposes a DNS failover mechanism, whereby modifying the DNS A-record associations of the Public IP address during the start of the recovery process, traffic can be directed to the recovered site with a minimal delay.

Based on the insights and data gathered during and after the evaluation phase of the proposed design, the tool created with Ansible and Terraform was found to be functional, performant and cost efficient within the known limits and expectations set by legacy disaster recovery practices.

**Keywords**: disaster recovery, infrastructure-as-code, web application

# Table of contents

## List of Tables



## List of Figures

# 1 Introduction

## 1.1 Motivation

To succeed in today's highly competitive IT software services market, companies need to be able to prove their capability of meeting stringent requirements with a solution offering that is both cost efficient and modern. Due to a steady growth seen in the adoption rate of public cloud services within the Finnish business landscape, as visualized in Figure 1, service providers are also being pushed towards adopting strategies which involve shifting their operations infrastructure from in-house, private datacentres to those managed by public cloud service providers or CSPs such as AWS, Azure and Google Cloud. Considering infrastructure costs and sales margins, this transitional movement has levelled the playing field, as the transparent pricing structure listed by most CSPs has left little to no room for manoeuvring and competition between companies that offer their products and services on top of these CSPs platforms.

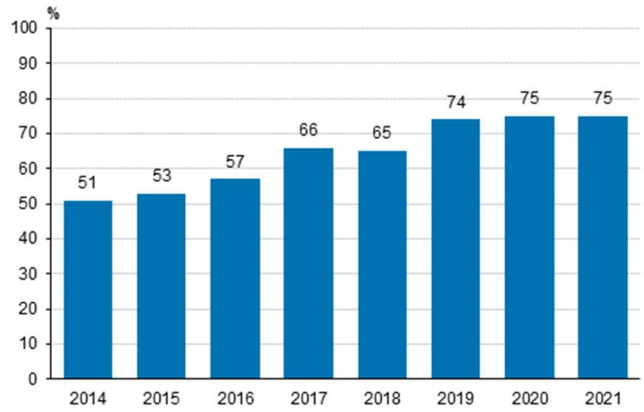


Figure 1 Usage of paid cloud services in Finnish companies [1]

Given this market setting, the question then arises: how can one compete or differentiate within this operational landscape? Suggestions for this were searched from the "State of DevOps 2019" report [2], published by Google, which through interviews with multiple companies operating in the cloud, aims to distinguish the efforts prevalent in the practices of organizations proven for excellence in technology delivery and powerful business outcomes. Within this report, the practice of disaster recovery testing emerged as one key differentiator of interest.

Falling into the area of infrastructure management, disaster recovery has been identified as a natural target for finding efficiencies when operating in the cloud, mostly due to the latest developments seen in modern infrastructure management practices and the constantly growing spectrum of different cloud services. Legacy, industry established disaster recovery approaches are known for impeding a relatively high additional expenditure, thus limiting the usage of such mechanisms only to the most business-critical IT systems. However, with the emergence of Infrastructure-as-Code (IaC) practices, this paradigm can now be challenged. As IaC allows the handling of infrastructure management in a textual format, the premise of having a fallback disaster recovery system created 'out of thin air' in the case of a disaster, instead of having it operational 24x7 in parallel with the system it protects, makes up for a compelling area to search for efficiencies.

## 1.2 Research questions and objectives

In this thesis the design and implementation process of a novel disaster recovery tool, which makes use of industry established Infrastructure-as-Code practices, is presented. This tool will be tailor-made to recover a specific web application or *target application*, and its respective platform architecture, the design and selection process of which will also be covered as a part of this thesis. As an objective for the design and implementation phase, the answer for the following research question is explored:

- **RQ1:** What aspects should be considered when designing an IaC -based disaster recovery for a web application?

For evaluating the implemented disaster recovery tool, the target web application and its platform infrastructure is configured to an operational state, after which an artificial disaster is generated, rendering the application unusable. The implemented disaster recovery tool is then started, with the aim of completely recovering the applications operation.

The main objective of the testing phase of this thesis is to gain confidence in the design of the tool and to evaluate the tools efficacy and cost implications that result from the its usage. Selected metrics will be followed, and the accumulated data from multiple rounds of test execution will be used to assess the overall time performance, reliability, and stability of the disaster recovery tool. Further, this data is used to assess whether the assumptions made in the following research questions hold:

- **RQ2:** Is the practice of Infrastructure-as-Code mature enough for it to be used in the recovery of a web applications operation after a disastrous event?
- **RQ3:** Does the cost efficiency expectation hold when comparing an IaC -based approach to industry established disaster recovery approaches?

In addition to the full recovery functionality, a testability functionality that allows testing the disaster recovery tools usage in parallel without affecting the target application state, is incorporated as a part of the tool's repertoire. This added functionality will also be tested as a part of the overall evaluation of the tool's operation.

The overarching objective of this thesis is to develop insight within the subject matter by answering the previously listed three research questions. This insight can then be used by the commissioner company to support the solving of related business problems, namely, to assist in the assessment of whether IaC -based disaster recovery techniques could be used as a part of future business offerings.

## 1.3 Research method

As a framework for answering the previously listed research questions, this thesis uses the design science approach, whereby an invention is proposed, its expected outcome is predicted and consequently evaluated.

Following the design science methodology, the disaster recovery tool, and its design act as the proposed invention. For setting the expectation, the predicted outcome for the designed tool is that it can be used for the complete recovery of a web application, with confidence in its operation and cost-efficiency. Whether the tool meets these expectations will then be evaluated during the verification phase of this thesis.

As a result, this thesis is expected to produce two distinct artifacts: 1) a design for a IaC -based web application disaster recovery tool, and 2) associated source code for a tool created based on this design. In addition, the expectation is that during the design and verification process, multiple 'lessons learned' bits of information are gathered, which can provide for those considering future designs and improvements to such a tool.

## 1.4 Thesis structure

The rest of the thesis is organized as follows.

In chapter two, necessary background information of the three central concepts and their respective terminology are introduced, with the purpose of providing the reader of this thesis a solid understanding of the subject domain. As no pre-existing web application was provided for the purpose of this thesis, against which the disaster recovery system could be run, the selection process of a target web application and the design of its underlying platform infrastructure is explained in Chapter 3. Given the actual design of the disaster recovery tool will be tailor-made against the target application, this chapter lays the groundwork for the actual design phase considerations, which are presented in Chapter 4. Then, in Chapter 5 the focus is put on the actual implementation work and the effort of evaluating the selected designs efficacy. Insight into the results of the testing phase and design aspects are summarized in Chapter 6, leaving Chapter 7 for concluding remarks.

## 1.5 Source code

At the time of the completion of this thesis, all related source code artifacts were released under the CC-BY 4.0 license and made publicly accessible through the following Github repository:

- https://github.com/tommi-tuomisto/utu-msc-drac

# 2 Central Concepts

## 2.1 Disaster Recovery

A disaster is an unexpected, major outage that renders a service provided by a computer system or application unusable to a point where basic fault tolerance and high-availability means are inefficient for recovery [3]. Further, if left unresolved, a disastrous event can be expected to cause severe damage to businesses and organizations relying on the affected services. Disastrous events can include both natural phenomena, such as earthquakes, floods, and fires, along with less tractable events, such as security breaches, human-error in configuration and administration, among others [3].

Disaster recovery is a mechanism specifically designed to dampen the effect a disastrous event might have on a system or service, ideally restoring its functionality [3] and assuring complete service continuity. Disaster recovery planning is typically driven as a part of a business continuity plan [12], or BCP, where different mechanisms for avoiding outages and tolerating failures within an organization are described. Whereas fault protection mechanisms exist to avoid outages happening at all to the extent that they are not visible even to the end users, and high-availability measures target to prevent minor outages, disaster recovery is considered as a solution aimed solely for resolving major outages and preventing long duration breaches in service availability [3].

### 2.1.1 Motivations for Disaster Recovery Planning

Implementing a DRP or disaster recovery plan is usually preceded by careful examination and weighing of an IT systems criticality from the perspectives of the organization or individuals using the system, the organization providing the system and the industry under which the system operates [11]. When considering whether DRP should be part of an organization's business continuity processes, the need for assessing both internal and external motivators was identified[1].

Internal motivators address the direct causality of a possible disaster and its effects to the day-to-day operations of an organization. For example, if most of a business organization's employees are reliant on a software tool or service for all work-related tasks, having this

[1] The categorization of motivators to internal and external ones was done based on the author's own observation after reading relevant literature.

specific system non-operational would in effect halt the productivity of the organization. For organizations that provide such business-critical systems to other organizations, protecting the most critical part of the services architecture is another example of an internal driver towards DRP. IT software services are typically bound by contractual agreements known as SLAs or service level agreements, which guarantee a level of service availability to the service consumer and define a reimbursement penalty scheme in case of a breach of this guarantee. Thus, if a service provider were not to implement DRP, a disastrous event could have a direct impact on the service provider organization through monetary losses accumulated from SLA penalties.

External motivators on the other hand, are those imposed by parties which otherwise might not play a part in the fulfilment or use of a service itself. These include mainly standardization authorities specific to certain industries and the judicial systems of a country the organization operates in. Failing to adhere to the requirements or recommendations of such authorities can have varying impacts, ranging from being unable to participate in bids as a service provider to being fined or made accountable for negligence through legal proceedings.

As an example, through its Guidelines on ICT Security and Risk Management, the EBA or European Banking Authority states that all EBA compliant financial institutes must incorporate adequate disaster recovery methods and continuity management processes to ensure a stable business environment and to avoid negative impacts on the financial system as a whole [5]. Continuing with the financial industry, the Finnish Financial Supervisory Authority similarly states, “There must be recovery plans for IT systems describing how each IT system can be reactivated in the event of a serious disruption or catastrophe” [7].

Further, VAHTI, the information security governance entity of the Finnish central government describes the framework of requirements that should be adhered to by providers participating in bids for government IT systems. In its “ICT -varautumisen vaatimukset” VAHTI requires, that in order to ensure fast recovery, service disruptions are to be prepared for. Further, VAHTI requires that for services rated from the level “Basic” and above, the core functions of an organization, recovery processes for such systems need to be described. [13]

Although such regulations and guidelines are not necessarily enforced by law, as is the case with FIN-FSA [8], in 247 a § “Laki sähköisen viestinnän palveluista” of the Finnish law states that in its risk management processes an organization operating in the web must take into

account “2) the management of information security threats and disruptions”, “3) management of business continuity” and “5) adherence to international standards”, if providing marketplace, search engine or cloud services [9].

### 2.1.2 RPO and RTO

The ongoing event of an IT system or application being unavailable for its end users due to unexpected events is called an unplanned outage [14]. To decrease the impact of such an outage, the general goal of a recovery method of any kind, is to minimize its duration by returning the service to an available state as soon as possible. As a ‘best practice’ with disaster recovery planning, an acceptable level for outage duration and its impact on potential data loss is set through two objectives: the Recovery Time Objective (RTO) and the Recovery Point Objective (RPO).

RTO sets the maximum available time within which the effort of service recovery from a disastrous event must be completed. Calculating RTO starts from the point of time when the disaster is noticed, thus excluding the promptness of possible monitoring tools or related processes from the measurement. The point in time where complete service availability is again achieved is consequently used to establish whether the RTO was successfully met. [3]

RPO, or Recovery Point Objective, is a measure for quantifying the allowed data loss in a unit of time, typically days or hours. RPO sets the maximum interval at which data should be made recoverable for a disaster recovery system. For example, given an RPO of 20 hours, if a disastrous event occurs at the point in time X, the recovered data must be at most from the point of time X - 20 hours. [3]

When measuring actual results, RPA for Recovery Point Actual and RTA for Recovery Time Actual can be used [6]. Continuing with the previous example, if a disastrous event occurs at the point in time X and the latest data recovery point exists at X - 5 hours, the RPA is 5 hours. With the original RPO of 20 hours, this translates to the objective being met with a 15-hour margin.

### 2.1.3 Established solutions

For implementing a working disaster recovery system, robustness and redundancy mechanisms are typically incorporated at the scale of complete IT systems. Three approaches for such system architecture design exist: the shared system, the hot standby system, and the

cold standby system approach [3]. Since physical disasters cannot be excluded from the set of possible disastrous events, both approaches are generally complemented with a varying degree of geo-local separation between the subsystems that make up the complete, recoverable architecture [3][10].

With the shared system approach, instead of relying on a single system to provide the service in question, the workload generated by the system's end users is shared with and serviced by multiple identical subsystems. Each individual subsystem acts independent of the availability of the other subsystems, i.e., having one subsystem non-operational is by design not allowed to impact the state of others, and thus the overall service operation.

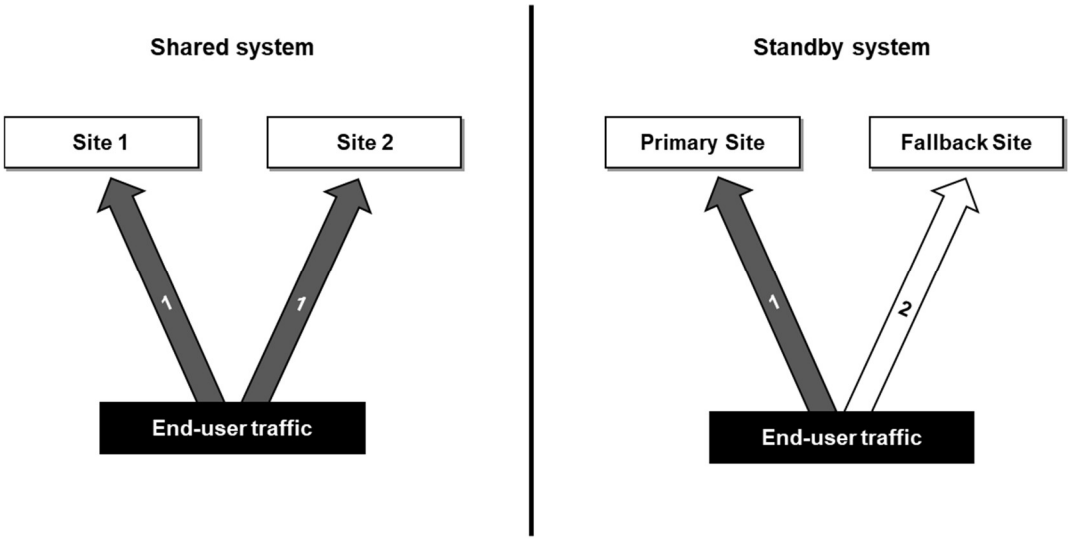


Figure 2 User traffic behavior, shared system vs. standby system

With the standby system approach, initially all traffic is directed against the primary system. If a disastrous event were to render the primary system unusable, the service would ‘fall back’ to a standby system. Once the fallback operation is complete, the end-user traffic would be directed against it, as is shown in Figure 2 with the 1 →2 transition. A standby system can be further categorized as ‘hot’ or ‘cold’. A hot standby system is constantly operational and run parallel to the primary system, while a cold standby system is brought up only in the case of an actual disaster. The cold standby approach is at times even further split into ‘warm’ or ‘cold’ solutions, with the distinction that while both can be offline during the moment of disaster, a warm system has been prepared further than a completely cold one. For example, a warm disaster recovery system might have the appropriate software installed while a cold system might include only the physical hardware, requiring all software installations to be

included in the disaster recovery process. This distinction however is not precise and can vary depending on the context and scope of operations included in the disaster recovery process.

### 2.1.4 Disaster recovery testing

The purpose of disaster recovery testing is to gain confidence in the created disaster recovery plan and process, and to increase the level of certainty that the process will work in an event of a disaster [11]. Creating a disaster recovery plan without including a proper testing process might lead to a situation where a change in any part of the plan, were it personnel changes responsible for managing the process or external dependencies which the actual disaster recovery mechanism relies on, might go unnoticed, resulting in severe consequences in the event of an actual disaster.

Going through this categorization, a paper test refers to a process simply consisting of reviewing the disaster recovery plan as it is documented. A walkthrough test is a similar review of a written document, with the distinction that a group of experts rather than single individual is gathered to go through each step of the process and provide comments on possible errors and improvements. As opposed to these two testing types, parallel testing is the first type of testing activity that includes performing the activities against a parallel system, meaning that the primary system is kept unaffected by the testing activities. Finally, the cutover test is used to test an actual workload shift from the primary system to the recovery system. This type of testing carries the highest amount of risk, as it can cause a real impact to the primary systems operation. [11]

In addition to selecting between testing types, a decision about testing frequency is also important. Naturally, the more often a disaster recovery process is tested the more confidence in its efficacy can be gained.

### 2.1.5 Cost vs. objectives trade-off

“A design for disaster recovery is a compromise between financial expenditure and the redundancy achieved”.[3]

As disastrous events are expected to be rare [3], over allocating resources for recovery measures may incur unnecessary costs in the long run. Thus, as the level of compromise fitting to each organization and use case is unique, emphasis should be put on weighing the implications of a disaster against the expenditure of the possible counteractive designs.

The previously mentioned solution architectures have inherent differences between achievable RTO and RPO. Regardless of the approach, all disaster recovery systems must ensure some level of state synchronization between the systems which together form the disaster recoverable architecture [3]. Because the shared system architecture requires a continuous synchronization of state, in order for all subsystems to be able to service the end user, it's RTO and RPO can be expected to meet the most stringent of demands. Similar expectation holds with the hot primary-fallback approach. Since additional steps for preparing the system are required with a cold fallback model, an increase in recovery time is expected.

From a cost perspective however, the hot fallback solution can be considered the most expensive one [10]. Having a continuously operating secondary system that is not serving any user generated load, can result in a near doubling of costs when compared to a non-recoverable architecture. With a cold fallback system this expenditure can be lowered, by keeping the system offline during normal operations.

For shared systems the cost effect is not as obvious. In some cases, a shared system might not be designed for true redundancy, i.e., losing one subsystem is allowed to affect some users. This can happen for example with a system that is optimized for a certain level of load, which is then shared evenly between each subsystem. In such a case, if one subsystem were to be rendered unusable, users serviced by this subsystem would experience a break in service until the affected subsystem is fixed or until the average load decreases to a level which can again be handled by the remaining subsystems. With true redundancy, the load would be constantly met with an amount of subsystems sufficient to service all users in the event of one subsystem being rendered non-operational.

## 2.2 Server Infrastructure

From an Information technology standpoint, the term infrastructure refers to the underlying and supporting hardware and software component stack that enables the execution of IT services and solutions. The components within this stack range from web server software to servers and networking hardware devices, and datacentre facilities [15].

When a web server operating within an operating system sends an HTTP response to a client, the response packet must be encapsulated to lower layer protocol frames - such as TCP, IP, and Ethernet frames - for the packet to be able to traverse between two computers. This encapsulation cannot be handled by any single software component. A web server relies on

both the operating system it is running on, and a network interface managed by the operating system, for preparing the packet for network traversal. Thus, both the operating system and the network interface are examples of infrastructure components that enable a web server's operation.

The basic operational nature of the client-server model of a web server demands the computational capacity to be higher on a server than on a client. To meet this demand, server infrastructure generally consists of specialized hardware and software, designed to fulfil specific computational tasks efficiently, consistently, and reliably. For this reason, when using the term infrastructure in the following chapters, we will refer solely to components with close relation to server-side functionality.

Further, we narrow the scope of infrastructure to consist only of components that can be manageable by the applications proprietor, leaving out for example wider area transit networking devices, or infrastructure facilities governed solely by datacentre operators, internet service providers or cloud service providers, along with end-user devices.

### 2.2.1 Operating systems

An operating system is a software infrastructure component [17] that operates between application software and computer hardware. Its main purpose is to manage the underlying hardware devices and orchestrate user and program access to all shared resources, such as CPU, memory, storage, and networking capacity. Operating systems also handle the actual execution of application software and provide various error handling mechanisms for malfunctioning devices and applications [16].

Operating systems designed for server application workloads are of ‘Multi-user and Multiprocessing’ type, where the OSs time-sharing-based scheduling capability allows multiple users to utilize the system simultaneously while also allowing multiple applications to be run at the same time [16]. The market of web server operating systems of this type is dominated by two distinct families of operating system software: UNIX and Windows. The most notable member of the UNIX family is Linux, which currently operates as a platform for up to ~50% of all UNIX web servers and over 30% of all web servers [224-1].

### 2.2.2 Physical infrastructure

Physical infrastructure includes devices that provide capacity and capability in three distinct areas of information technology: compute, storage, and networking [19].

For computing, the most common hardware device is the server computer, where the hardware is made of powerful computers, built with high-end, industrial-grade components [20]. Given the nature of their modifiability, server computers can host a wide range of different workloads, from web server applications to databases and game servers.

Storage infrastructure devices are dedicated for a specific task, in this case storing and distributing data. With storage systems, attributes such as high-capacity data storage and throughput can be achieved by including built-in storage infrastructure subcomponents such as large arrays of hard disk or solid-state drives, and special purpose disk controllers. [21]

Lastly, networking hardware devices, the most common of which are listed in Table 1, are responsible for managing a shared communications medium used for connecting different hardware devices together and allowing them and their respective applications to communicate with each other.

Table 1 Common networking devices and their use cases [18].

| | |
|---|---|
| **Switch** | Connects multiple devices to form a shared computer network. |
| **Router** | Routes traffic between different networks |
| **Firewall** | Security device for monitoring and controlling network traffic |
| **Repeater** | Amplifies an incoming signal and rebroadcasts is |

For all the three main categories: compute, storage, and networking; infrastructure components can be extended to also include attachable devices. For example, networking hardware also includes attachable networking components, such as Network Interface Cards, or NICs, that can be used for extending the networking capabilities of other hardware devices. [18]

### 2.2.3 Virtualized infrastructure

Virtualization refers to a mechanism of indirection, where a consumer of a resource is presented with a virtual rather than a physical representation of the consumed resource [22]. The achieved benefit of virtualization with computing systems is that it enables sharing a single resource or device, be it a networking device or processor, between multiple consumers, as if each individual consumer were the sole consumer of the actual resource. This is achieved with the introduction of an intermediary software layer, i.e., the virtualization layer, which interfaces with the actual resource and relays and shares access to it through a programmatic abstraction [24] for each individual encapsulated consumer workload. Thus, from the perspective of an individual consumer, all inputs, outputs, and behaviour [23] of the virtualized resource resemble those provided by a non-virtualized one.

In terms of server infrastructure, the three main types of virtualization are:

1. system (platform or server) virtualization,
2. network virtualization, and
3. storage virtualization. [23]

System virtualization refers to the consolidation of physical servers and the isolation of guest operating systems to encapsulations known as Virtual Machines, or VMs. The sharing of the underlying computing resources provided by a physical server is managed by a specialized software called a Virtual Machine Monitor (VMM), also known as a hypervisor. To allow for effective virtualization, a hypervisor is expected to provide a complete representation of actual computing interfaces for a VM, including CPU instructions, virtual memory, and I/O devices [23].

With network virtualization, the shared physical resource is the networking medium and its packet-forwarding capabilities, which are provided by the underlying combination of switching, routing and access control software and hardware [26]. Unlike with system virtualization, the offering for network virtualization techniques is more varied, based on the level of detail, encapsulation and configurability exposed to the consumers of the virtualized resource [25]. In other words, different network virtualization techniques exist and are selected for use depending on how much control over the virtualized network a network tenant is expected to require. With virtual local area networks or VLANs for example,

separate network tenants can utilize the same physical network, as if each tenant had its own physical layer of networking infrastructure available. However, with VLANs the control over this network from a single tenant's perspective is minimal, as for example restrictions to IP addressing selections are imposed, to avoid both overlapping addresses between tenants and consequential routing issues. To allow for more detailed segmentation, the next level option would be to utilize a virtual extensible LAN or VXLAN, a more enhanced network virtualization technique which addresses most limitations imposed by VLAN [27]. Continuing this trend, the latest advancement in network virtualization has been made available by SDNs or software defined networks, where the control logic is separated from the physical devices and exposed for the tenants to utilize. Contrary to VLANs, SDNs allow each tenant a more flexible set of controls over the shared resource, in terms of isolation from other tenants, and control over addressing and network topology [25], resulting in a more comprehensive set of abstractions over the physical capabilities of a network.

Storage virtualization allows the abstraction of physical storage hardware and its management, with the purpose of further commoditizing storage capacity allocation as is done with physical computing resources such as CPU and memory. Instead of having dedicated physical disks for a physical or virtual server, virtual storage can be allocated from a pool of heterogeneous storage and presented to its consumer as a single storage device. [28]

### 2.2.4 Extending infrastructure with cloud services

Because of the advancements made in effective resource sharing and tenant isolation, virtualization has played a critical role in the development of new computing service models, such as IaaS [29]. With IaaS, the previously mentioned virtualization of compute, storage and networking capability is provided as a service, allowing users to utilize these functionalities without the need of owning and operating their own hardware infrastructure. With cloud services these functionalities are provided through the web, on an on-demand basis [29].

However, the current offering from major cloud service providers such as AWS, Azure, Google Cloud and others has since far exceeded the feature set of traditional IaaS, with for example many commonly used application-level workloads such as databases, DNS, and NAT being made available as managed services. From an infrastructure management perspective, this has brought application-level workloads closer to the practices of provisioning lower-level infrastructure. Instead of requiring users of the IaaS platform to install and configure these applications separately, with the managed service model higher

level service functions such as the ones mentioned above can be deployed as plug-and-play components, in conjunction with lower-level infrastructure components such as VMs.

In addition to typical application-level workloads, cloud services are also being used to bundle a close set of features together, with the goal of allowing a user to form a more complete system on one go, instead of having to deploy each individual component of a system separately. For example, a typical cloud-based computer network consists of multiple VMs, with firewalls dictating the type of traffic that is allowed to traverse to and from each VM. Instead of deploying each VM individually and continuing then with the deployment of a separate firewall, some cloud service providers couple the firewall settings with the VM settings [31][32], thus abstracting the actual firewall from the system architecture and bundling its configuration together with the VM.

## 2.3 Infrastructure-as-Code

Rather than a specific technology or tool, infrastructure as code or IaC is a set of infrastructure management practices [35], where the same rigor of application code development is applied to infrastructure provisioning, and all configurations related to creating and maintaining infrastructure is defined in a declarative way and stored in a source control system [34].

The main advantage of using IaC is realized when managing a system consisting of a large number of infrastructure components [35]. Whereas systems of only a couple of virtual machines can be managed with manual work through a web or command line interface, increasing complexity and size in system architecture will quickly grow the management workload to unsustainable levels. Contrary to legacy system administration operations, IaC also introduces speed and repeatability, allowing for faster setup times and higher frequency changes. With the growing trend of DevOps and continuous delivery practices, the expectation toward the agility of infrastructure management has followed [33][36], thus putting emphasis on the importance of such features.

### 2.3.1 Orchestration of desired state

The management workflow used with IaC can be split into two distinct phases: managing state and orchestration. The former contributes the code part of the IaC abbreviation, with which the desired state of the system under management is described in scripting language or

notation format. This description is then, either through manual or scheduled invocation, executed against the target service, e.g., an IaaS platform, by a software program called an orchestrator.[35]

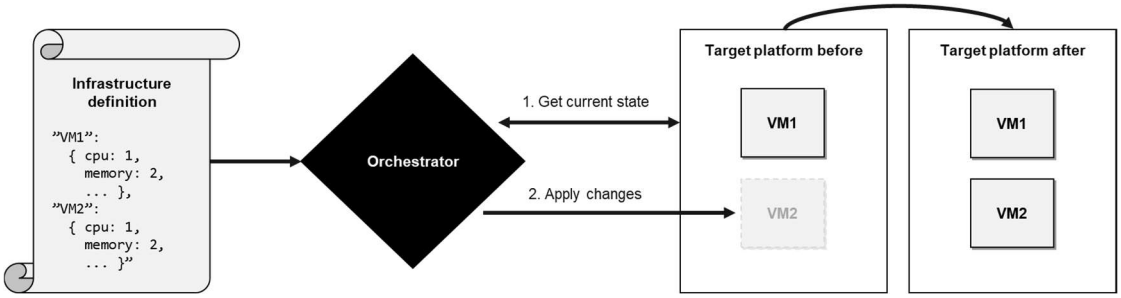


Figure 3 Example of orchestration operation

As shown in Figure 3, on a general level the primary functions of an orchestrator include the discovery of the actual state of the system, translation of the desired state into actionable tasks and finally, execution of tasks with the aim of matching the actual state with the desired one.

To achieve repeatability with orchestration, a level of idempotency is required to assure that despite multiple executions, the same descriptions will yield the same result [37]. Action will be taken only for the logical parts that differ, whereas if the desired state matches the current state, no action is required.

### 2.3.2 Orchestration software characteristics

The current offering regarding orchestration software includes both proprietary and general-purpose tooling, with each tool introducing variation in the way the previously explained operational flow is achieved in practice.

While some tools use standardized notation formats such as JSON, YAML or XML for defining the desired state, or are provided as software libraries with built-in support for popular programming languages [36], some have created their own, domain specific language or DSL. [38]

Secondly, the orchestrator can consist of a single entity or multiple entities. An example of multi-entity orchestration is the master-agent model, where the orchestration workload is split between agents, programs usually installed locally within the managed component themselves [38] and masters, central programs that are kept separate from the managed components. With this model, agents can be said to handle the actual execution of the management tasks, while

masters handle the higher-level orchestration, e.g., managing the state description files and delegating them to the agents.

Third, orchestration tools are either imperative or declarative when it comes to the level of granularity, importance of order and interdependence of individual actions within the state definitions. With a procedural approach, the state definition is a step-by-step guide on how to set up the desired state, with each consequent step being reliant on the success of those previously executed. With an imperative approach, the end-state is simply declared, leaving the procedure that is required to achieve the state for the orchestrator to handle. [38]

Fourth, handling changes to the desired state can be done either with a mutable or immutable routine. A mutable orchestrator can select the appropriate procedure from a set of possibilities for applying the change, usually with the purpose of causing minimal disruption to the targeted component during the change. With an immutable approach, no updating or patching to existing infrastructure is done [35]. As an example, when updating an operating system to a new version, a mutable orchestrator would only apply the changes that differ between the old version and the new one, while an immutable orchestrator would reinstall the whole operating system, for it to match the desired version.

Fifth, differences exist in feature coverage and applicability of the orchestrator when it comes to the management of the different layers of infrastructure and various cloud services [39] [40]. While overlap on coverage and features exist, most individual IaC tools are, or have originally been considered particularly useful for managing a specific subcategory of components. Also as mentioned, some tools are proprietary, i.e., they allow the management of only specific cloud services components, or vendor specific products. General purpose tools on the other hand can be used for managing components from various vendors.

Finally, orchestration can rely on different communication protocols for passing the instructions, with most common ones being SSH for achieving remote connectivity and HTTP for passing instructions to web APIs [35].

### 2.3.3 Version control of state definitions

Version control systems are software programs used for handling multiple versions of a piece of information [41]. Also known as source code management, version control is a well-established best practice within the software development industry for storing source code and configuration files. Although version control cannot be categorized as an obligatory

component in IaC to a similar degree as state management and orchestration, as orchestration tools can be used without version control, the added benefit of using common version control features such as reverting changes, auditing and collaborating [42], makes incorporating such a mechanism into the infrastructure management lifecycle beneficial.

With IaC, state definitions are typically stored as files. How changes made to these files are delegated for the orchestrator and at which point should the version control system step in, is not predefined and allows for flexibility in terms of the extent at which do organizations responsible for managing these files want to collaborate. For instance, one organization might allow making changes to the infrastructure definitions and consequently executing them, only publishing the changes to version control once the target system has been deemed functional. Another approach might be to require publishing the changes to version control for approval, before passing them for execution.

The most popular version control program is Git, with an adoption rate of over 70% compared to other version control software, based on statistics of open-source repositories on the Internet [43]. On top of Git's open-source features, several popular services such as Github and GitLab have been built, enabling the use of complementary features such as visual navigation and collaboration within Git repositories, syntax proofing during publishing, among others.

# 3 Target web application and infrastructure

Given the purpose of this thesis is to test the recovery of the web application operation, not the operation of the web application itself, early on in the process it was identified [47] that adding to the complexity of the web application operation and its underlying platform would not add any value to the outcome of the disaster recovery testing, but rather add to the difficulty of the IaC setup, while incurring unnecessary costs. Thus, all decisions in the below chapters were mirrored against these considerations, resulting in an architecture that is kept simple, yet representative of a real-life implementation while also cutting on redundant mechanisms that only serve as adding to the complexity and cost overhead.

## 3.1 Target web application

### 3.1.1 Web applications vs. other web content

Web applications can be considered a subset of all software applications, with the distinction that they are reliant on a computer network, and their functionality is constructed with web specific technologies such as HTML, CSS and JavaScript, and rendered by a web browser for the end user to utilize.

A web application operates under the client-server computing model [45], where communication between the client and the server is done over HTTP; an application layer protocol of the TCP/IP protocol suite. HTTP itself operates on a request-response communication model, where a specific set of request methods for the client and responses for the server are defined. A client -- typically a web browser located in a personal computer, smartphone, or other similarly capable end user device – is responsible for translating the user's actions to HTTP requests and sending them to a web server, which processes the request and responds to the end-user in the form of a HTTP response.

While web applications cover a wide array of use cases, from online shopping portals and personal mailboxes to search engines and corporate intranets, they do not represent all the content that is shared within the public or private networks they operate on. In consequence, as the previously mentioned web technologies can be used for the creation of web pages of any type, it is important to distinguish a web application specifically in contrast to other web content.

The most essential functionality of a web application relates to its ability to be interactive and present dynamically tailored content. This is achieved by allowing the application to modify its response shown to the end user, i.e., the final web page presentation, based on request parameters, tracked user behavior, and security considerations. [45]

Dynamic web pages may allow the user to affect the presented response. For instance, it is very common for larger websites to provide its user a possibility to do an internal search query against all text-based information stored within the website, allowing the user to quickly find what one is looking for. The search result thus depends solely on the user-given input and might vary between users. Further, for some web applications a user might be required to provide personal security credentials, to access data that is purposefully of limited visibility to others. Such security functionality not only gives a highly personalized experience but also allows the applications creator to better manage and track who is using the application and how. As such, web applications can be considered as tools that in addition to information storage and sharing, also allow the modification and manipulation of information and how it is presented, in ways that can provide added value to both the applications users and its providers.

### 3.1.2 Liferay

Deciding which web application to use as a target for disaster recovery was done based on discussions with the representative of the thesis commissioner [47]. Summarizing these discussions, the following three criterion were considered to have most weight on the decision:

1. The web application has an existing market,
2. it is relevant to the commissioning company’s business and
3. it is available for non-commercial use

While these criteria were initially evaluated against many possible candidates, Liferay was eventually selected as the target web application. Liferay is an enterprise portal developed by Liferay Inc., a company which since its inception has grown its portfolio to offer a suite of digital software solutions, such as customer self-service portals and B2B commerce solutions [46]. In Finland, Liferay has a significant market, especially in the public sector, with 33 % of Finnish governmental organizations having chosen Liferay as their publishing application

[48], making Liferay a highly relevant choice also from the thesis commissioner perspective. Further, from the point of view of this thesis, having chosen an application of such significance, allows the work produced to be more applicable to real world use cases.

For its enterprise portal, Liferay comes in two editions: 1) Liferay DXP or Digital Experience Platform (formerly known as Liferay EE or Enterprise Edition) and 2) Liferay CE or Community Edition. The difference between these two editions exists in the support model, where to better serve enterprise customers, the commercial DXP platform is offered with a greater level of support and guarantees when compared to the CE platform, which can be used free-of-charge [49]. Thus, from a DR testing perspective, as no technical reason was found to select one over the other, the CE was selected.

## 3.2 Multi-tier web application architecture

Liferay is an exemplary application for following a multi-tier architecture [47], a well-established software application architecture, where the workload processing is organized into distinct computing tiers [50]. With Liferay, these tiers can be organized as follows:

1. Presentation tier
2. Application tier
3. Data + search tier

The presentation tier represents the UI or User Interface logic, which for web applications such as Liferay is built with CSS, JavaScript, and HTML, and is executed by the web browser. The application tier is the backend of the application, executed by an application server program. The application tier is responsible for processing the data gathered from the presentation tier and for executing data retrieval and modification queries towards the data tier [50]. The Liferay backend is built in the Java programming language, with supported Java application servers including JBoss, Tomcat, Websphere, Weblogic and Wildfly [51]. The data tier is responsible for storing and managing the information that is processed by the application. Within the database tier, both relational database management systems or RDBMS, and non-relational or NoSQL databases can be utilized. For RDBMS, Liferay supports MySQL, MariaDB, PostgreSQL and Oracle database software [53]. In addition to relational databases, Liferay utilizes Elasticsearch as a full-text search engine [52], which is used for converting searchable database entities in the RDBMS to JSON documents. After

having been indexed by Elasticsearch, retrieving information through these document objects is faster as compared to directly querying the same information from a relational database [54].

### 3.2.1 AWS for networking, compute, and storage

The selected platform for the web application infrastructure is AWS or Amazon Web Services. AWS provides all the basics of computing infrastructure as a service, virtual machines, storage, networking, and security along with an offering of over 200 different featured cloud services [57] ranging from managed databases, serverless computing, container orchestration platforms, etc.

From a computing, network and storage infrastructure allocation perspective, various strategies can be used for supporting the execution of a multi-tier application. For example, one could have the application server and the database share the same operating system along with the virtual server. However, such an approach is counter-intuitive considering the benefits the multi-tier architecture aims to achieve, namely the services of each tier being customizable and optimizable without impacting the other tiers in the process. In helping customers succeed with these architectural decisions, Liferay publishes a best practice guide for deploying the application on a cloud services platform in a manner that information security, performance, fault tolerance and scalability measures are considered [55] [56].

The selected architecture is partly based on the recommendations made in “Deploying Liferay Digital Experience Platform in Amazon Web Services” -guide, with some exceptions made due to pricing and scaling factors. For example, instead of running a managed database via the AWS RDS service, a database is configured to run on its own virtual machine. Second, as the load to be generated towards the application will be low, the need for a load balancer was considered minimal and thus left out of the design.

For forming the multi-tier architecture in AWS, the following services will be utilized: VPC, EC2 and EBS. Together, these services form the architecture shown in Figure 4.

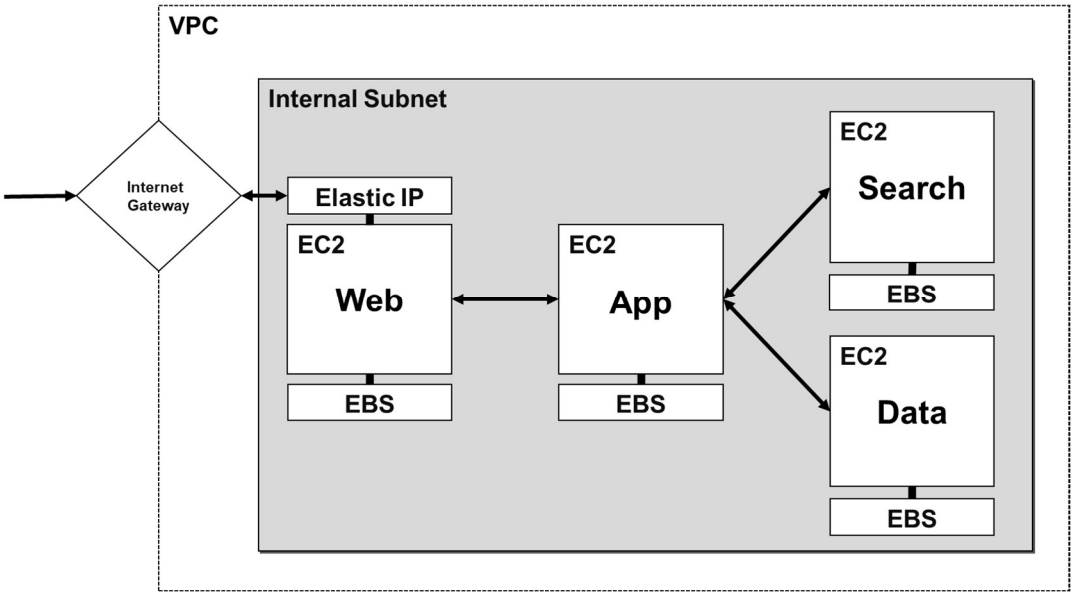


Figure 4 Selected architecture in AWS

VPC or a Virtual Private Cloud is a networking service used for forming a private and dedicated network for each individual AWS account. With VPC one can manage how EC2 instances, the AWS equivalent of virtual machines, are networked by offering a feature set similar to what is available in traditional data center network management. In the selected architecture, VPC is used for creating a private subnet which allows connectivity between the EC2 instances. Route tables are configured to allow routing traffic from the EC2 instances to the Internet, via the Internet Gateway, a AWS managed VPC component that is placed on the edge of the VPC [61]. To have the EC2 instances accessible from the public internet, public IP addresses known as Elastic IPs in AWS are assigned to the EC2 instances [58]. Securing the environment is done with the use of Security Groups, a virtual firewall feature of a VPC that controls inbound (ingress) and outbound (egress) traffic for an EC2 instance. With security groups one can define an IP range from which connections to a specific port can be made. Security groups can be used for example to restrict the access to the SSH port, allowing remote connectivity to happen only from known IP addresses. Security group rules are stateful, i.e., inbound restrictions will not apply to incoming responses, which are requested by the EC2 instance and vice versa [59].

Within the VPC, EC2 instances form the computing platform of the architecture. EC2 instances are offered in various types and sizes, with some types being suitable for general purpose computing while others are optimized for a specific workload such as big data processing or high-performance computing. Within each instance type, a range of differently

sized instances in terms of memory, vCPU, network bandwidth and storage bandwidth capacity are offered. [60]

For storage, no external services are utilized, but rather local volumes created with the EBS or Elastic Block Storage service are attached directly to the EC2 instances. These block-level storage devices can be used as one would use a physical hard drive [62]. Given that only one EBS volume is provisioned per EC2 instance, this volume acts as the root volume, and is used for storing both the operating system and application data. Similar as with EC2 instances, EBS volumes are offered in different types, such as SSDs ranging from general purpose devices to ones providing low latency, along with HDDs that can be used for data warehousing and throughput favored workloads. For the selected architecture, EBS volume sizes were determined on the basis of what is required at minimum for supporting the installation of the software binaries of the

### 3.2.2 AMIs and the Amazon Linux 2 operating system

The role of the operating system selection was estimated significant from two standpoints: a) the OS is manageable with IaC and b) the OS is supported by Liferay. In AWS, selection of the operating system which will be installed to the EC2 instances, is based on the available AMIs or Amazon Machine Images listed in the AWS Marketplace. An AMI is a template for the root volume of the instance that describes which operating system to use and which EBS volumes to make available for the OS [63]. In the Liferay architecture recommendations, choice of AMI is suggested to be done based on the provided compatibility matrix released for each Liferay version separately. Liferay 7.2 compatible operating systems include all major providers, ranging from Linux distributions from Red Hat and Oracle to Microsoft's Windows Servers [51]. However, utilizing these would require a license purchase, in order to have access to the application package repositories that are needed for installing the required application-level binaries needed at each tier. Thus, from the free Linux based operating systems, CentOS, Ubuntu, Debian, Alpine Linux and Amazon Linux 2 were left for consideration. Since no real distinction between these distributions could be made in terms of IaC management capabilities, Amazon Linux 2, the OS created by AWS was selected.

In AWS, the AMI images are referenced through an AMI ID, an identifier unique to each operating system in an AWS region. For obtaining the AMI ID, the AWS Marketplace was searched for the selected operating system and for the specified AWS regions. Obtaining this

information is important, as all AMI related IaC management actions rely on having the correct ID.

## 3.3 Role of the URL in web application operation

A web application is accessed through a web browser, by inputting a URL or Uniform Resource Locator to the address bar of the browser. A URL consists of multiple different parts, each serving a unique purpose in the delivery chain from request to response.

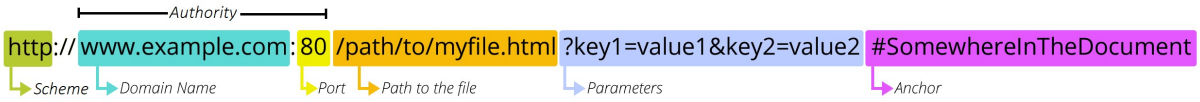


Figure 5 “What is a URL?” by Mozilla Contributors [76]

From the perspective of this thesis, the fields of most interest are ‘Scheme’ and ‘Domain Name’ fields shown in Figure 5, as they are dictated by design decisions made at the web application infrastructure level, rather than the web application software level, to which the latter three fall into. In the following chapters, the importance of these two fields is explained and a design for their implementation is selected.

### 3.3.1 DNS

While a web server can be accessed by sending requests to its public IP address, domain names are usually preferred, as they are more convenient to use from an end-user perspective. In most cases[2], using domain names instead of IP addresses cannot be done without utilizing DNS.

When a user requests a web server resource through the browser by using a domain name, the browser first sends the domain name to a DNS system to be resolved into an IP address. The IP address provided as a response can then be used to process the actual HTTP request.

[2] It is possible to use a “hosts” file as an alternative method for domain name resolution, where hostname to IP -mappings are manually configured to a plain-text file stored in the end-user’s device.

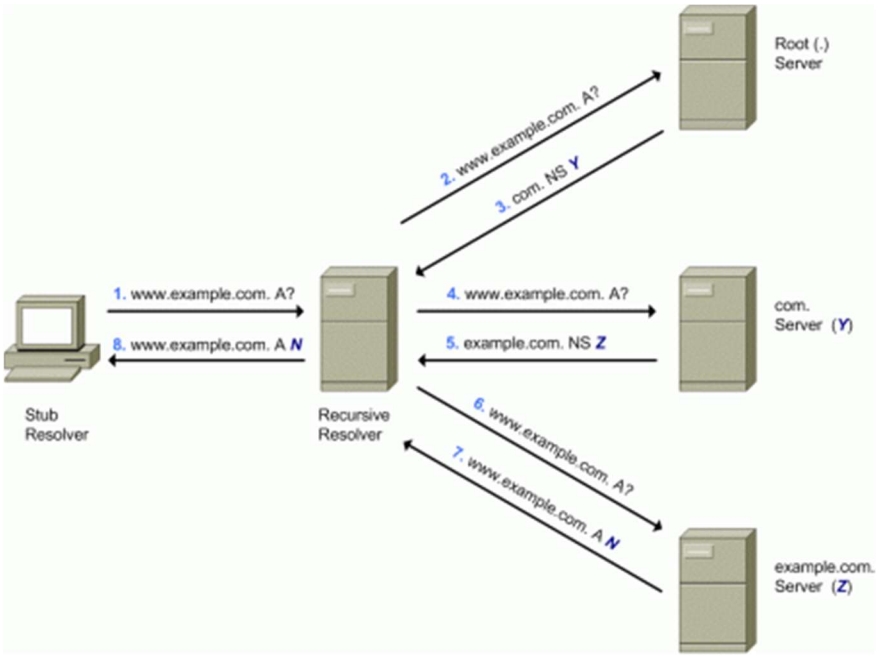


Figure 6 DNS architecture and its operational flow [75]

A DNS system responsible for resolving domains which are accessible through the public Internet, consists of resolvers and of three specific types of servers [75], as shown in Figure 6. Within this system architecture, a client makes one DNS query to a Recursive Resolver (Figure 6, Step 1), which will then handle all remaining queries (Figure 6, Steps 2-6). While a root server holds the information of the suitable top-level domain, or TLD servers (Figure 6, Server Y), a TLD server consequently holds the information of the Authoritative server (Figure 6, Server Z). The authoritative server is then queried for the actual domain-to-IP translation (Figure 6, Step 6), after which the answer is returned to the client.

### 3.3.2 Cloudflare as selected external DNS service provider

The decision whether to operate your own Authoritative DNS or use a DNS service, was reflected against the number of domains under management. Since the target infrastructure only hosts a single application, which will be accessed through a single domain, having one server for this purpose was considered inefficient both in terms of added cost and computing capacity overhead that would follow. Further, having the Authoritative DNS server operate within the same infrastructure as the web server infrastructure, would in the case of a disaster result in the loss of both web application and DNS service. Given these constraints, Cloudflare was selected as the service provider, through which DNS functions would be managed.

Cloudflare is a company that operates one of the biggest privately owned networks in the world, the capabilities of which are provided to Cloudflare customers in the form of services such as DNS, DDoS protection, content delivery, and caching among other performance enhancing functionalities [65]. Cloudflare has a free tier service, allowing individual users to utilize a limited set of Cloudflare features, among them DNS, free of charge. Most importantly however, Cloudflare can be managed with IaC [64].

By selecting Cloudflare, the provided DNS services used as a part of the disaster recovery process is considered external and unaffected by any disastrous scenario the web application infrastructure might experience.

### 3.3.3 TLS and HTTPS

In Figure 5, the ‘Scheme’ field dictates the protocol to be used with the request. For modern web applications that are expected to process sensitive information such as credit card numbers and health records, handling communications between a client and a server using HTTP is considered insufficient from an information security perspective. This is because HTTP does not include functionality that would protect the message data from eavesdropping or tampering when sent over a computer network [73]. For this reason, most web traffic today [74] is encapsulated within an encryption protocol known as TLS. The combination of HTTP and TLS is commonly known as HTTPS. Short for Transport Layer Security, TLS provides a way for establishing a private and reliable channel for secure communication of application data between two parties, which in the context of a web application translates to the client and server.

As is the case with Liferay, handling this encryption is not done by the web application software itself but rather by a separate web server application, which in the selected three-tier architecture acts as a reverse proxy between the application server and the client. As such, the web server can be considered as the point-of-termination for the TLS encryption process [66].

Setting up a web server for TLS capability requires the generation of an asymmetric public-private keypair and an accompanying digital certificate, which is used for proving the ownership of the public key to clients connecting to the server.

### 3.3.4 Let's Encrypt for automated management of certificates

The conventional process for certificate management has been to buy a certificate from a certificate authority or CA, an organization or company that validates the identities of companies, individuals, websites, and email addresses [67]. For ordering a certificate, a CSR or certificate signing request is sent to the CA. A CSR specifies the domain name for which the certificate is to be created, and certain organizational information that are used for identifying the validity of requesting organization. Once the certificates are received, they are deployed in a manner that makes them available for the web server to utilize. However, many automated processes that eliminate the need for such a manual approach exist. For example, in AWS, certificates can be handled by the Certificate Manager service or ACM, through which one can both request a certificate and deploy them automatically to other AWS resources [68]. Another automated approach is made available by Let's Encrypt, a CA service provided for free by the Internet Security Research Group or ISRG. [69] Automating certificate handling with Let's Encrypt is accomplished by running a certificate management agent on the web server [70]. In the selected three-tiered architecture, this translates to installing certbot, a software tool used for automatically deploying Let's Encrypt certificates for supported web server applications [71], on the Amazon Linux operating system hosting the web server.

From the perspective of IaC or disaster recovery, selecting between automated or non-automated processes was considered non-essential from a certificate acquisition point-of-view, while having the deployment for usage was considered a hard requirement. Since ordering a certificate from a conventional certificate authority incurs a relatively high cost, starting from 100 $ upwards for a single certificate, it was ruled out in favor of cheaper options. Since there would be no AWS provided TLS termination service within the selected architecture, ACM was not considered as useful, although its pricing structure would allow deploying a single certificate nearly free of charge [72]. As Let's Encrypt is a totally free service and installing and configuring certbot to the Amazon Linux EC2 instances was estimated achievable with the current state of IaC techniques, utilizing this service was selected for certificate management.

# 4 Design of disaster recovery system

## 4.1 Primary-fallback with cold standby

To assure the assumptions made in the research questions are tested, especially in terms of cost efficiency, the primary-fallback operational model for the disaster recovery system was selected. Translating the primary-fallback model into AWS based infrastructure, separation was done on the level of AWS accounts, i.e., for both the primary and fallback sites, a separate AWS account is created. Geo-local separation between these sites is achieved by selecting two different AWS regions within the European geographic area, under which the VPCs of the two sites will operate. Other than having the account pre-created and made accessible for the IaC tooling, no other configurations will be done at the fallback site. Thus, the baseline in terms of infrastructure readiness on the moment of disaster can be categorized as ‘cold’.

On top of this cold baseline, the IaC DR tool is expected to build a near identical duplicate of the primary system to the fallback site. “To establish IT functionality on the disaster recovery system, the system must have all applications, necessary configurations, and all data”. To complete this description to meet the expectations set for the DR tool, the designed recovery system must also have all server infrastructure: virtual machines, storage devices, networking, and security.

### 4.1.1 DNS Failover and TTL

The reason why both sites cannot be completely identical, lies in the Public IP addressing scheme. As AWS is the proprietor of the public IP addresses assigned as Elastic IPs to the EC2 instances within a VPC, a certain IP address cannot be allocated at will to an AWS resource, but rather each Elastic IP assignment results in a random IP selection from a pool of AWS owned IP addresses. To deal with this limitation, the DNS Failover mechanism is utilized.

With DNS Failover, traffic going to a malfunctioning server can be directed or ‘failed over’ to a functioning server, simply by changing the domain-to-IP translation. How fast this mechanism can take action depends on the TTL or Time-To-Live parameter of the DNS specification of a given DNS entry, held by the Authoritative DNS server. The TTL describes to the clients how long a DNS translation can be used until it should be considered invalid,

prompting for a new DNS query to the resolver. For example, once a DNS query has been made, the result of the query is saved as a temporary record in the client system, along with the TTL associated with the query. If a new query against the same domain name is done within the TTL period, no actual query against the DNS resolver is required and the temporary record is used instead. Once the TTL runs out, the next effort for resolving the IP address of the same domain will require a new query against a DNS Resolver. Thus, in case of disaster recovery, the lower the TTL value, the faster the new IP address will be taken to use by clients.

To achieve DNS Failover, the process requires a two-step approach: 1) associate the fallback web server EC2 instance with an Elastic IP, and 2) modify the associated Elastic IP address to the A-record of the primary domain used for accessing the web application. A DNS *A record* is the primary type of DNS record, used for indicating the IP address of a specific domain [77]. In the selected architecture these associations are managed through Cloudflare, where the default TTL value is set to 5 minutes [78].

### 4.1.2 Parallel testability considerations

To make the disaster recovery process testable in a parallel manner, the main mechanism to utilize is also DNS. To allow the fallback site application to be run in conjunction with the primary site application, neither can affect the others' operation. Thus, possible overlaps between two separately running, but similarly configured sites need to be considered.

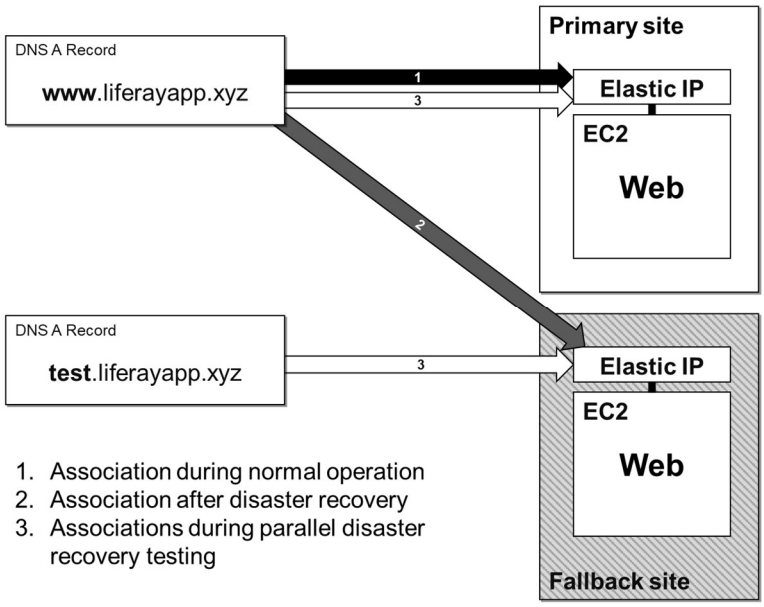


Figure 7 DNS failover mechanism between two sites

As these two sites are designed to be run in separate VPCs and AWS regions, networking conflicts can be ruled out, leaving the only overlap in the domain record. To have both web applications accessible through a domain name, a mechanism for creating separate domain names for each site must be implemented. For this purpose, another subdomain name of the primary TLD must be assigned to the A record association pointing to the fallback site's web server's Elastic IP. This results in the scenario depicted in Figure 7, where three different DNS record associations must be supported by the disaster recovery process.

## 4.2 Recovery in two phases

Because disaster recovery processes are tightly linked with state synchronization, i.e., a complete recovery restores the data from a recovery point, having the tool capable of only doing disaster recovery was considered a disadvantage. Thus, a decision to split the process into two distinct phases was made:

1. Recovery of infrastructure components, application binaries and stateless configurations

2. Recovery of application data and stateful configurations

The motivation for this split originates from the difference between an operational application and a recovered application. An operational application is one that has been properly configured and is available for use. A recovered application is one which is both operational and available for use, while also enriched with data and configurations taken from the latest available recovery point. With the cold fallback model, the disaster recovery process must include a mechanism which handles the synchronization of data and stateful configurations, from the recovery point to the fallback web application.

For this to be achievable in practice, the 2nd phase of recovery is designed as an addon functionality, meaning the same IaC definitions that are used for orchestrating the setup of the stateless infrastructure components are utilized with an additional set of definitions invoked in the case of a disaster. This guarantees that the state of the two sites match the IaC definitions at any given moment, allowing for a *single source of truth,* the importance of which will be explained in chapter 4.1.2. Further, this approach allows the infrastructure of the primary site to be managed separate from the disaster recovery process, enabling a process that resembles actual real-life infrastructure management processes, where stateless infrastructure components can be modified without affecting the actual state of the application and its data.

Selecting this approach made it also easier to test the tools usage, as the same tool could be used for creating an operational application on top of which the user data could be inputted through manual actions in the UI.

Whether or not an IaC managed component falls into the 1st or 2nd phase was evaluated based on whether actions done at the web application level affect the state of the component. This was considered for each component, categorized by its architectural tier in Table 2.

Table 2 Phase selection based on stateful data and configurations location.

| | Server Infrastructure | Application binaries | Application configuration | Application data | Phase 1 | Phase 2 |
|---|---|---|---|---|---|---|
| *Web* | No | No | No | No | **Yes** | No |
| *Application* | No | No | **Yes** | **Yes** | **Yes** | **Yes** |
| *Database* | No | No | No | **Yes** | **Yes** | **Yes** |
| *Search* | No | No | No | **Yes** | **Yes** | No * |

As seen in Table 2, the web tier of the multi-tier Liferay setup is a completely stateless tier from the point of view of the web application, as any actions, be it modifications, additions or deletions at the application user interface have no impact on how the web server operates, how its configured or what type of data is stored to its filesystem. The application tier on the other hand has both types of components. For example, the storage of all multimedia files that can be shared in Liferay are within the selected system architecture, stored to the EBS volume that is attached to the Application tier EC2 instance. The application server also holds configuration files that can be manipulated both from the command line, or directly from the web application itself. Further, all non-multimedia data is stored by Liferay into the database tier. While the database configurations themselves do not change from end user action, the actual database is expected to be continuously evolving as users manipulate the Liferay portal through the UI.

Finally, the search tier holds the search index data. In case of Elasticsearch, the search index is stored as a complex JSON object, that is continuously evolving as more content is added to the portal. The search tier was omitted from 2nd phase and the backup and recovery process, as the search index can be recreated from the existing application and database data through a

process known as ‘reindexing’. If the index is large, which can be the case if the application itself handles a large amount of data, executing a re-index might not always be the most efficient approach. However, this will not be the case with the target application, as it will only host a minimal amount of data for testing purposes.

The only exception to the categorization shown in table KJH is log files. Typically, all applications are configured to write a log file to the file system on top of which they operate, to allow for example troubleshooting possible application issues. This is the case also with the applications of each tier described here. Contents within these log files are dependent on user action. For example, when a user makes a request for a particular web server, this action is usually logged. However, the content written to these log files do not affect state or the functionality of the application itself. Thus, from a disaster recovery point, log files are considered insignificant and will be omitted from the recovery process.

## 4.3 Handling data backups

As covered in chapter 2.1.2, RPO is the objective used for defining the maximum interval between recovery points of data. While in some disaster recovery approaches this interval may translate to direct state synchronization between the primary and fallback sites, for the selected cold fallback model this is not the case. Rather, the RPO dictates the backup interval, i.e., how often data from the primary system is copied to an external location, from where the disaster recovery process can then, once the disaster recovery process is initiated, retrieve the data and apply it to the fallback site.

How this backup procedure is handled was considered non-essential given scope of this thesis. Further, as no RPO related testing will be included, no periodical backup procedure needs to be planned for.

Regardless of the interval, some form of backup procedure must be created to make the data available at the point of disaster. The selected approach for handling backups was to have a simple script made in *Bash*, a Unix shell and command language, and have it executed prior to starting the disaster recovery process, making the data and stateful configurations of the application and database tiers available for the IaC DR tool to utilize.

## 4.4 IaC tooling and usage

### 4.4.1 Terraform and Ansible for orchestration

Terraform and Ansible were selected as the IaC tools for implementing the infrastructure management and disaster recovery orchestration.

Terraform is a general-purpose tool that can be used for managing a wide range of different infrastructure resources, from low level computing, storage, and networking components to cloud and SaaS service features [79]. With Terraform, infrastructure is defined in a domain specific Hashicorp Configuration Language (HCL). These configurations are then translated by the Terraform application in to requests towards the API of the platform on which the managed components are run on top of. Whether or not an infrastructure platform or service is supported by Terraform depends on whether a *provider* has been made available for it. A Terraform provider is an interface that provides the translations from HCL notation to API calls. A provider can be considered analogous to a software library, which once imported, allows the utilization of the functions and features included in the library. Similarly, a provider is taken into use in the HCL notation.

In Figure 8, the basic orchestration workflow of Terraform is described. Once the configurations have been completed (*Write* phase) the first thing to do is run an *Init* in the Terraform project, which downloads the defined providers to the local environment, allowing their further utilization. The *Init* step needs to be done only once within the Terraform project. Then, the execution starts with the *Plan* phase, where the changes that will be made are first showcased in a dry-run fashion. The Plan phase allows one to do a final review, before the changes are actualized in the *Apply* phase, after which Terraform works to have the definition match the actual state of the environment. How this state is managed is explained in more detail in the next chapter. Finally, Terraform can be configured to output selected information as a result for the execution. [80]

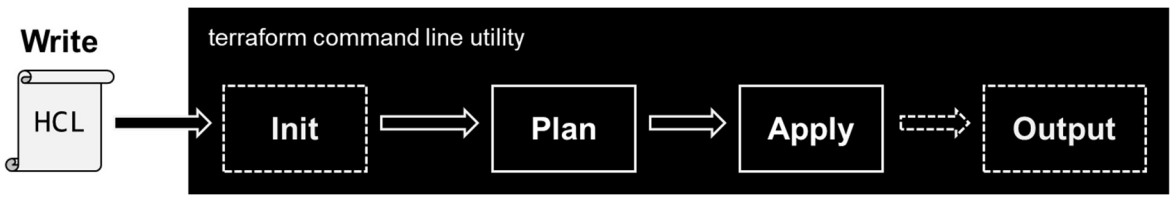


Figure 8 Basic Terraform workflow

Ansible is another general-purpose tool that can be used for cloud service provisioning, configuration management, application deployment among a plethora of other IaC related tasks. The primary mode of operation for Ansible is to manage the resources, by connecting to them with the SSH remote management protocol. In addition to SSH, the managed resource must have support for one of the following languages. While Terraform is a declarative tool, Ansible requires a procedural approach when writing the instruction for infrastructure management. In Ansible, these instructions are called *playbooks* and they are written in a domain specific language similar to the YAML notation format. Each playbook consists of individual *plays*, small idempotent tasks that are designed to part of the playbooks overall purpose. Further, *plays* can be made into *roles*, self-contained orchestration functions that can be shared between different playbooks. [83]

The selection of these two tools was based on the achieved coverage, i.e., in combination, the selected tools had to be capable of managing the whole scope of the web application architecture. Further, both Ansible and Terraform are open-source software tools that can be utilized by individual and enterprise users, free-of-charge. Both AWS and Cloudflare have their respective Terraform providers, making them manageable by Terraform, while Amazon Linux 2 includes a pre-installed Python and SSH packages, thus satisfying the requirement for Ansible management.

### 4.4.2 Handling orchestration state

As introduced in chapter 2.3.1 one of the main tasks of an IaC orchestrator includes handling the state of the infrastructure under management. If the orchestrator were not to have a complete and current understanding of this state, the outcome of consequent orchestrator actions could not be reliably predicted.

In Terraform the state is handled through a file known as the *state file*, a plain-text file that stores the infrastructure state in JSON format. The state file has the following functions:

1) acts as a database for mapping the definitions to actual resources
2) includes metadata, such as resource dependencies, depency order etc.
3) acts as a cache, by storing attribute values for all resources listed in the state file. [81]

The state file can be stored either locally or remotely. With the local approach, the file is stored directly within the Terraform project, which also holds the actual infrastructure

definitions. This approach works in situations where there is only a single individual contributor, or all infrastructure management actions are initiated from a central location. However, as using version control software such as Git is a common practice when practicing IaC, the resulting collaborative effort typically leads to a situation where each contributor has their own version of the IaC codebase. In such a case it is important to have a single source of truth for the actual state, which all contributors can reference reliably. It is not recommended to store the state file to a version control system, due to it containing sensitive information about the target infrastructure.

As only a single contributor will work on the IaC codebase related to the disaster recovery tool described in this thesis, handling the statefile is done with the local approach.

### 4.4.3 State and variable separation with Terraform Workspaces

Having the ability to execute three different scenarios with the same infrastructure configurations, can be achieved with the use of Workspaces, a functionality of Terraform which allows multiple states to be associated with a shared infrastructure configuration [82]. Further, using Workspaces allowed the use of different sets of variables. With this approach, the created infrastructure configurations can be made to act as templates. For those components within the configurations that require a differentiation from the other sites, the value will be dictated through a variable which is set in a workspace specific configuration, while those components that remain the same regardless of the site can be shared by each workspace and left static in the template configuration. The splitting of variables is important, as they are used as the sole mechanism for differentiating the orchestration actions between the different sites.

### 4.4.4 Combined execution

Enabling the execution of the whole disaster recovery execution pipeline through a single Ansible command was achieved by using a community Terraform plugin, which allows for the execution of Terraform commands through Ansible playbooks.

Selecting this approach from possible alternatives was done based on the following estimated benefits:

1) Enables the use of Ansibles built-in execution report that can be utilized for assessing the success of each individual action of the process. Error handling.

2) Enables the use of Ansibles built-in execution report for collecting execution time data.

Thus, the resulting flow of execution is as shown in Figure 9.


Figure 9 Ansible playbook structure and division to two phases

The process is initiated with the ansible-playbook command. Specifying between different scenarios is done by prompting the user in the beginning of the playbook to select from three different alternatives, through a string identifier. This identifier is then used by Ansible to initiate Terraform in the respective workspace. Similarly, the identifier is used in the later stages of the execution, to ensure the application configurations match the created infrastructure.

# 5 Implementation and verification

## 5.1 Preparations

### 5.1.1 Cloud service accounts creation

The actual implementation work of the disaster recovery testing environment started with the creation of the two AWS accounts, which would be used for hosting the primary and fallback virtual machines, or EC2 instances, as they are called in AWS. The only hard requirement for setting up an account in AWS relates to the billing information. One must be able to provide a working credit card from which the costs accumulated from AWS resource usage will be billed monthly. Once the accounts were created, resources in them could at first be managed only through the console. As the selected infrastructure as code techniques require programmatic access, the only step after account creation was to create a set of keys for the account user, which could then be used by Terraform to authenticate against the AWS API.

In addition to AWS, an account for Cloudflare was created. To have the Liferay application IP address resolve to a domain name through Cloudflare, the service user must either move an existing domain name under the user's ownership to Cloudflare or purchase a new domain through the Cloudflare portal. As no existing domain was available for this purpose, the latter approach was selected. The domain ‘liferayapp.xyz’ was selected and purchased for a minimum, 1-year term. Similar to the AWS setup, in order to allow the selected IaC tooling to manage the DNS records, an API token for authentication purposes was created.

Once the initial account configurations with API keys and tokens were set up, all consequent management activities for both AWS and Cloudflare accounts were done with IaC through the development environment.

### 5.1.2 Development environment setup

An important part of an IaC system is the local environment used for the development and execution of the IaC tooling. For a project developed by a single person, separating these two tasks to individual environments was deemed an unnecessary division. Thus, the same environment was used for both developing the IaC code and executing it against the remote cloud services.

Fedora 34 Workstation, an open-source linux distribution, was selected as the operating system for the local environment. The selection was done on the basis of supported platforms for the required IaC executables, Ansible and Terraform. In addition, Fedora includes built-in development tools, including SSH for remote connectivity, vim for file editing and git for version control. Tools that were not included are shown in Table 3. These were installed separately, by following vendor provided instructions.

Table 3 Installed IaC related binaries and their versions.

| Binary | Version | Purpose |
|---|---|---|
| AWS CLI | 2.3.0 | Provides the authentication method used by the AWS provider in Terraform. |
| Terraform | 1.0.9 | AWS and Cloudflare orchestration |
| Ansible | 2.9.25 | Disaster recovery tool orchestration |

To ensure the availability of the latest features, the most recent stable versions at the time of development for each binary were targeted.

Following established IaC practices, a dedicated repository for IaC definition storage and version control purposes was created to the authors Github account. Github is a publicly available internet service that allows individuals to store their version-controlled data, free of charge. The connection and authentication between the local environment and Github was done with SSH. For this purpose, a dedicated ssh keypair was created, from which the public part of the key was placed in the Github accounts settings, while the local environment SSH was configured to use the private part of the keypair when connecting to the Github server. Once the connection was established, all consequently created IaC definitions and configurations were periodically uploaded, or ‘pushed’, to the remote Github repository.

To allow Ansible to connect to the remote virtual machines, an additional and dedicated SSH key was created for this purpose.

Finally, the folder structure created for all infrastructure definitions followed the recommended structure of an Ansible project, with additional folders made for the terraform definitions, application installation binaries and backup files. See Appendix 1 for the full directory structure and content.

## 5.2 Fully automated application setup

### 5.2.1 Terraform configurations

Within the created folder structure, the first task was to create the Terraform project configurations under the ‘terraform/’ directory, which would be used for building the server infrastructure on top of AWS and Cloudflare. This included utilizing terraforms documentation for mapping out the required IaC definitions for the selected architecture in HCL. While not required by terraform, each logical segment, be it DNS, EC2, network or security setup was separated to its own definition file. In Figure 10, the configuration of a single EC2 instance with HCL is shown. In addition to HCL, the setup of EC2 instances required the creation of a *user data* script, a small Bash script executed automatically by AWS during the setup of an EC2 instance. This script was used for placing the public part of the SSH key into the Amazon Linux operating system, which would later be used by Ansible for managing the system.

```
resource "aws_instance" "search" {
  ami                         = var.al2_ami_version
  instance_type               = "t3.small"
  associate_public_ip_address = true
  private_ip                  = "10.0.0.12"
  subnet_id                   = aws_subnet.private.id
  vpc_security_group_ids      = [aws_security_group.allow_default.id, aws_security_group.allow_se.id]
  user_data                   = file("${path.module}/shell/init_ssh.sh")
  root_block_device {
    volume_size = 8
  }
  tags = {
    Name = "liferay-tier3-se"
  }
}
```

Figure 10 Terraform HCL configuration for EC2 instance creation. Screenshot.

The server infrastructure setup was first done to one site, after which separate workspaces for primary, fallback and test sites were created. This allowed workspace specific state files to be used, which were stored under the ‘terraform/terraform.tfstate.d’ subdirectories. In addition to the workspaces, separate variable files were created for each workspace, and placed under the ‘terraform/tfvars/’ -directory.

```
environment = "drac-primary-ttu"
region = "eu-north-1"
cloudflare_email = "thesis.author@email.fi"
cloudflare_api_token = "123456789qwertyredacted"
cloudflare_zone_id = "qwerty123456789redacted"
cloudflare_domain = "liferayapp.xyz"
cloudflare_record_name = "www"
al2_ami_version = "ami-0d15082500b576303"
```

Figure 11 Terraform variables for a single site. Screenshot.

As shown in Figure 11, the variable file was made to include the AWS service account name, the AWS region name, along with the necessary Cloudflare authentication and DNS configurations. The AMI IDs had to be also specified in the variables, as the same ID doesn't apply for different regions in AWS. To have the testability functionality match the actual fallback disaster recovery process, both the test and fallback sites were configured to target the same AWS region. Having the primary site operate in the 'eu-north-1' (Stockholm) region and the fallback and test sites operate in the 'eu-central-1' (Frankfurt), meant that two different AMI IDs had to be defined.

### 5.2.2 Management access and internal connectivity

For a functioning web application, only the web server EC2 instance would require a public IP address. However, to ease the management of each tier, all EC2 instances were assigned a public Elastic IP address. For the web server instance, this IP would be used for both management and web application access, while for the application server, database and search instances, the only function of the public IP address was to allow SSH based management access to Ansible. For securing the environment with security group definitions, SSH access was allowed only from a specific IP address, while all HTTPS traffic to the web server was allowed. Further, traffic from one EC2 instance to another was limited only to the specific ports required by the applications operation: Port 80 of the application server instance was allowed to be accessed by the web server instance, while ports 5432 of the database server and 9200 of the Elasticsearch server were allowed to be accessed by the application server.

It should be noted that having a publicly available interface attached to each instance might not be the best approach for securing such an environment. Another approach could be to use a bastion host [86], that would act as a single point of entry, through which the other instances could be accessed. In this case, remote connections and consequent management actions to

the remote instances without public IPs would require an indirect connectivity approach, for example by utilizing the ‘jump host’ capability of SSH [87].

### 5.2.3 Executing Terraform within an Ansible playbook

For the rest of the setup the selected approach was to create a single Ansible playbook file that refers to individual roles in the ‘roles/’ directory, which are created for each contained step of the recovery process as described in Figure 9. To allow all phases of recovery to be executed through a single playbook, the terraform infrastructure management would have to be wrapped inside of the Ansible execution. This was made possible by using a Terraform community plugin created for Ansible. Installing the plugin to the local environment, made it possible to call terraform commands from Ansible playbooks and roles. The AWS and Cloudflare setup done by terraform would act as the first role in the disaster recovery playbook, with each consequent roles relying on the outcome of this setup.

The problem faced with this approach was that prior to the execution of the playbook, there would be no knowledge of which IP addresses will be assigned to the EC2 instances. This information is crucial, as all consequent application setup tasks rely on this information. Typically, Ansible is used by configuring an inventory file, which lists all the known hostnames and/or IP addresses of the server infrastructure under management. In this case, such an approach was not possible. The static inventory file was configured to only hold the ‘localhost’ hostname and SSH connectivity parameters. However, Ansible allows for the inventory to be dynamic, i.e., a host could be added to it while running the playbook. To make use of this possibility, Terraform was configured to output the assigned IP address variables post-execution. These variables would then be caught by the Ansible terraform plugin and added to the inventory. For starting the terraform role, the localhost target is used, as no other host yet exist in the dynamic inventory. At the end of the role, a check was created, to ensure the correct number of hosts were added to the inventory. In addition, a separate play for waiting for the EC2 instances to become ready was made. Readiness in this context meant that Ansible could access each EC2 instance added to the inventory with SSH.

### 5.2.4 Operating system and application configurations

Once SSH connectivity to EC2 instances was established, configuration plays regarding the operating system setup, application binary installations and configurations could be started.

For installing the required application software to the Amazon Linux operating system, three separate approaches were used:

1) Install the software binary through the operating systems own package manager,
2) copy the installation file from the local environment and execute it manually in the operating system, or
3) download the installer directly from a third-party

For the web and database tiers, all required software installations could be done through the package manager provided by the operating system. For Elasticsearch, the package manager could also be used, but a specific Elasticsearch repository had to be configured which also required installing a GPG public key to verify the authenticity of the repository source. For Liferay, no package repository existed through which the application could be installed. Thus, the required binary was downloaded from Liferay's website and copied to the application tier OS and then unarchived, after which the application was ready for taken into use.

To configure these application software programs to a required operational state, either a complete representation of the configuration file or a template configuration was used, and then copied from the local environment to the applications home directory by Ansible. For example, with the NGINX and Liferay configuration files, a templating functionality was required, since these configuration files included the assignment of the server FQDN within the configurations, which changed according to the targeted site, be it 'primary', 'fallback 'or 'test'.

For all tiers, the applications were configured to operate under *systemd*, a Linux 'daemon' which is commonly used for managing application processes. This allowed a simple way for starting, stopping or restarting the application, when needed. Systemd also has built-in support on Ansible, i.e., starting and stopping the application services could be reliably executed without installing external plugins or running tailor-made commands within the Ansible roles.

The final phase of the application setup was the integration role, where the last configurations assuring interoperation between all tiers were done, after which the Liferay application was started.

### 5.2.5 Changes to TLS Certificate handling due to Let's Encrypt rate limiting

The only issue faced with the application platform setup was noticed with TLS certificate handling while doing initial testing on the developed infrastructure code, where orchestration activities were executed multiple times. While the original plan was to install certbot to the web tier operating system, and have the certificates renewed for each round of execution, this could not be done in practice due to the Duplicate Certificate rate limit set by the Let's Encrypt service. This feature of the Let's Encrypt service limits the renewal of certificates subject to the same domain, to 5 times per week [84]. As the initial development phase testing included running the orchestration many times in a day, this limit was quickly met.

Due to this unexpected limitation, the installing of certbot and renewing the certificates with it was left out of the automated process. Instead, certbot was used only once during the development phase for requesting a wildcard certificate for the 'liferay.xyz' -domain. Using a wildcard certificate allowed subdomains such as 'www.liferay.xyz' and 'test.liferay.xyz' to utilize this same certificate for TLS encryption, as long as the TLD of the domain name matched the one in the received certificate. Once obtained, these certificates were then used for all development and testing activities. They were placed within the Ansible project structure, and as a part of the NGINX setup, copied from the local environment to the web server by an Ansible play. As the certificate bundle provided by Let's Encrypt also included the private key, all certificate related data were omitted from being published to version control.

### 5.2.6 Omitting sensitive information from version control

Prior to publishing files to an external, public version control repository, careful examination of the version-controlled files regarding information sensitivity had to be done, in order to avoid compromising private information such as usernames, authentication keys, IP addresses and other account details. Special attention was given to Terraform, which as a part of its execution, creates 'hidden' files that can be easily missed and uploaded by accident to version control.

To help avoid accidental publishing, Git provides a feature where filenames added to a special purpose file called '.gitignore' are filtered out from version control. This feature was also used for omitting large software or backup files from being published to Github. For all the files selected to be filtered out of version control, see Appendix 2.

## 5.3 Recovery functionality

### 5.3.1 Bash script for backup

To start with adding the recovery functionality, the backup script used for fetching the required configurations and data from the application and database tiers was first created. The script was written in Bash, mainly to differentiate it from the actual disaster recovery process.

Within the script backing up the application was started by obtaining the necessary IP addresses from the Terraform state files, which could then be used when issuing remote SSH commands to the application and database servers. Prior to creating the backups, the NGINX and Liferay processes were stopped, to ensure no modifications to the data could be made during the backup procedure, and to ensure that both the application data and the database data would be taken at a synchronized state.

For backing up the application tier, a guide published by Liferay on their website [85] provided the necessary information of which part of the Liferay home directory data should copied from the application tier OS. For fetching this data, the backup script was built to first archive the selected files and folders and then copy them to the local environment. Archiving the selected files served two purposes: 1) it allowed saving the UNIX directory ownership and execute permissions for each file and folder, reducing the need for creating such functionalities in the recovery process and 2) it allowed for faster data transfer times, as archiving the data on-site with an compression algorithm reduces the overall size of the data to be transferred from the remote machine to the local development environment and the EC2 instance.

For database backup, the script ran a single ‘pg_dumpall’ command, a command line utility provided with the PostgreSQL installation, which output all data included within the database to a single file. This file was then copied to the local environment.

### 5.3.2 Mirroring the backup procedure for recovery

The Ansible roles which would then be used within the disaster recovery playbook for actual recovery of the backed-up data to the application and database tiers were made to mirror the process of the backup script. First, the NGINX and Liferay services would be stopped, after which the recovery of the data would commence. Once all data was put in place, the last roles

of the whole disaster recovery playbook would ensure the services that were stopped in the beginning, were started back up again.

### 5.3.3 Phase separation and target site selection

The final task for completing the disaster recovery playbook was to incorporate two input mechanisms: a) one for separating the targeted site and b) one for defining whether to run a full recovery, or just create an operational application within the targeted site. For a) the implementation was a simple input prompt, set to be run as the first play of the whole playbook. Once executed, the playbook would not advance further before the targeted site is given as an input, at the beginning of the playbook execution. For b) the functionality was achieved by using ‘tags’, a Ansible functionality that allows selecting a subset of plays within a playbook for execution. All roles that were categorized as belonging to Phase 1, were given two tags: ‘operational’ and ‘recovery’, while roles that were required in full recovery were only given the ‘recovery’ tag. To take the functionality in to use, the ‘ansible-playbook’ command was invoked by specifying the ‘-t’ option, along with the desired tag name.

## 5.4 Verification

### 5.4.1 Areas of testing

For testing the created DR playbook, the target was to gather data that supports the evaluation of the tool’s efficacy in the following four areas:

1. Successful recovery
2. Performance
3. Reliability and stability
4. Testability

For evaluating successful recovery, the testing consisted of inspecting the web application through its UI and checking whether all modifications that had been done on top of the operational, primary application are present in the post-recovery fallback system. These modifications were done prior to running the backup script on the primary application, acting as the difference between an operational application and a recovered one. Finding these

modifications on the fallback site's post-recovery application, would result in a successful test outcome.

Performance evaluation consisted of a) measuring the recovery time of the application and b) measuring the execution time of each disaster recovery test round, by making use of Ansible's post execution report. Further, the execution times for each Ansible role were gathered, to allow the analysis of the most time-consuming areas of the IaC disaster recovery process. Recovery performance evaluation was done by an external monitoring tool, which was utilized for polling the availability of the web application URL with HTTP requests. The poll was executed in 1-minute intervals, with the monitoring tool expecting to find a specific 'search string' and a '200' status code from the HTTP response provided by the web server of the application. The search string was set to 'Welcome', a text string found in the landing page of an operational Liferay application. If the response for the poll fails to return anything, fails to find the search string from the response HTML code, or returns another response status code than the one expected, the site is considered unavailable. Gathering this data allowed for estimating the actual downtime and RTA of the application, as it would be experienced by an end-user, also including the effects of DNS TTL expiration to the downtime measurement.

Reliability was measured by executing the recovery process multiple times. Based on the initial performance findings during the implementation phase, one round of disaster recovery was estimated to last between 15 to 30 minutes. 10 rounds of test execution were considered to provide enough data. The outcome of this measurement was based on the percentage of success calculated after all rounds were completed. For example, if 7 rounds managed to successfully recover the application to the fallback site, the outcome would be 70 %. The time performance data provided by Ansible was also used for estimating the stability of the process, by calculating the average deviation between same sub-executions between test rounds. In addition, all other types of deviations between rounds, such as error message or performance variations were gathered for later analysis.

Finally, a single test round for executing the testability functionality of the disaster recovery tool was done. A successful outcome for this test required both sites to remain operational in parallel and accessible through their respective domain names, while having the same exact data one would expect with successful recovery.

### 5.4.2 Test execution

The flow of each test execution round for primary-fallback disaster recovery tests are shown in Table 4. With the single testability test round the ‘Disaster’ step was omitted and the ‘Recovery’ and ‘Cleanup’ steps were targeted against the ‘test’ site.

Table 4 Commands of a single test round, in order of execution.

| Step | Directory | Command | Target |
|---|---|---|---|
| Setup | / | ansible-playbook disaster_recovery.yml -i inventory -t operational | Primary |
| *Manual modifications to the web application through the UI* | | | |
| Backup | / | ./run_backup.sh | Primary |
| Disaster | terraform/ | terraform destroy -var-file=/tfvars/primary.tfvars | Primary |
| Recovery | / | ansible-playbook disaster_recovery.yml -i inventory -t recovery | Fallback |
| Cleanup | terraform/ | terraform destroy -var-file=/tfvars/fallback.tfvars | Fallback |

Going through this process, each round began with a clean installation of an operational application. This was done by running the disaster recovery playbook against the ‘primary’ site, along with the ‘operational’ tag. Once the application was successfully setup and available through the ‘https://www.liferayapp.xyz’ URL, the following modifications were done in the Liferay portal as a logged in user, to ensure data addition to all stateful tiers:

1. A blog site was created and made directly accessible from the landing page of Liferay
2. A blog entry was written, with a large image file as the header decorator, along with a few paragraphs of text.
3. A comment on the blog entry’s commenting field was made
4. Modifications in a user’s profile name and description were made
5. Login password was changed from the default one

After making these modifications, the backup script was executed, which stored the state of the modified application to the local development environment. Then, after having backed up the primary site application, an artificial disaster was rendered to the site by issuing the 'terraform destroy' command in the primary site workspace. This initiated the deletion of all resources that were included in the primary site statefile, effectively erasing its infrastructure and making the application unavailable. Following up the destroy, the same playbook that was run in the 'Setup' phase was then executed again, this time against the 'fallback' site and the 'recovery' tag. This initiated the actual disaster recovery execution, which recovered the modified state from the local backups to the fallback site application.

As a final step, the fallback site was cleaned up with the terraform destroy command to allow for the next execution to be issued against an empty environment.

### 5.4.3 Results

Successful recovery was achieved in all test rounds, resulting in 100 % overall reliability for the tool usage. Similarly, testability worked as expected, with both the primary and test applications providing the exact same content while remaining independent from each other and accessible through the 'https://www.liferayapp.xyz' and 'https://test.liferayapp.xyz' URLs, respectively.

As for the recovery time performance, the RTA as shown in Table 5 was on average 15 minutes, which translates to the total time of outage experienced by the application user. Both tool execution duration and the RTA remained stable throughout the test rounds, with only round 7 deviating with an approximately 2 minute above average tool execution duration.

Table 5 Recovery time performance results

| Round | Tool Execution Duration | Outage Start | Outage Stop | RTA | Δ (RTA – Tool Execution Duration) |
|---|---|---|---|---|---|
| 1 | 00.11.09 | 23.05.00 | 23.20.00 | 00:15:00 | 00:03:51 |
| 2 | 00.11.54 | 09.30.00 | 09.46.00 | 00:16:00 | 00:04:06 |
| 3 | 00.11.13 | 22.38.00 | 22.52.00 | 00:14:00 | 00:02:47 |
| 4 | 00.11.31 | 23.16.00 | 23.32.00 | 00:16:00 | 00:04:29 |

| Round | Tool Execution Duration | Outage Start | Outage Stop | RTA | ∆ (RTA – Tool Execution Duration) |
|---|---|---|---|---|---|
| 5 | 00.11.01 | 09.23.00 | 09.38.00 | 00:15:00 | 00:03:59 |
| 6 | 00.11.22 | 11.52.00 | 12.06.00 | 00:14:00 | 00:02:38 |
| 7 | 00:13.24 | 14.23.00 | 14.39.00 | 00:16:00 | 00:02:36 |
| 8 | 00.11.14 | 16.06.00 | 16.21.00 | 00:15:00 | 00:03:46 |
| 9 | 00.11.40 | 16.44.00 | 16.59.00 | 00:15:00 | 00:03:20 |
| 10 | 00.11.31 | 18.33.00 | 18.47.00 | 00:14:00 | 00:02:29 |
| Average | 00:11:36 | | | 00:15:00 | + 00:03:24 |

For evaluating the performance of the individual plays of the disaster recovery playbook, the initial report of Top 20 plays with highest execution time as reported by Ansible was refined to a report of averages, including only the Top 15 plays. Reducing the number from 20 to 15 was needed because of the deviation in plays included in the Top 20 report grew as the execution times became smaller. In other words, the reported list of plays changed most between placements 15 and 20. An example of the output given by Ansible after a single test round is shown in Appendix 3.

Table 6 Ansible Top 15 individual play average performance results.

| Phase | Ansible play | Average execution time (sec) | Average deviation (sec) | From total execution |
|---|---|---|---|---|
| 2 | Copy Liferay OSGi library from backup | 189,6 | 18,8 | 27,2 % |
| 1 | Copy Liferay installation file | 110,2 | 8,5 | 15,8 % |
| 1 | Run Terraform apply | 48,0 | 4,5 | 6,9 % |
| 2 | Restore OSGi | 38,6 | 0,2 | 5,5 % |
| 1 | Unarchive the installation file | 22,8 | 0,2 | 3,3 % |
| 1 | Install the latest version of PostgreSQL13 | 21,7 | 6,0 | 3,1 % |

| **Phase** | **Ansible play** | **Average execution time** (sec) | **Average deviation** (sec) | **From total execution** |
|---|---|---|---|---|
| 1 | Start the Elasticsearch service | 21,6 | 0,3 | 3,1 % |
| 1 | Wait 600 seconds for target connection to become reachable/usable | 20,7 | 5,5 | 3,0 % |
| 1 | Install the latest version of Elasticsearch from the repo | 19,1 | 1,6 | 2,7 % |
| 1 | Install OpenJDK | 18,4 | 0,6 | 2,7 % |
| 1 | Install the latest version of PostgreSQL13 server | 17,7 | 0,6 | 2,5 % |
| 2 | Restore from dump file | 9,8 | 1,5 | 1,4 % |
| 1 | Install required analysis-kuromoji plugin | 8,8 | 0,5 | 1,3 % |
| 1 | Install required analysis-icu plugin | 8,4 | 0,5 | 1,2 % |
| 1 | Install required analysis-smartcn plugin | 8,4 | 0,4 | 1,2 % |
| Top 15 from total execution time | | | | 81,0 % |

The Top 15 plays took on average 81 % of the total execution time of the whole disaster recovery playbook, with 19 % of plays falling under the 8,4 second execution time threshold.

As for the stability of the execution throughout test rounds, the average time deviation for each individual Ansible play in the Top 15 listing shown in Table 6 was below 10 %, with the most absolute deviation found in plays with the highest average execution time.

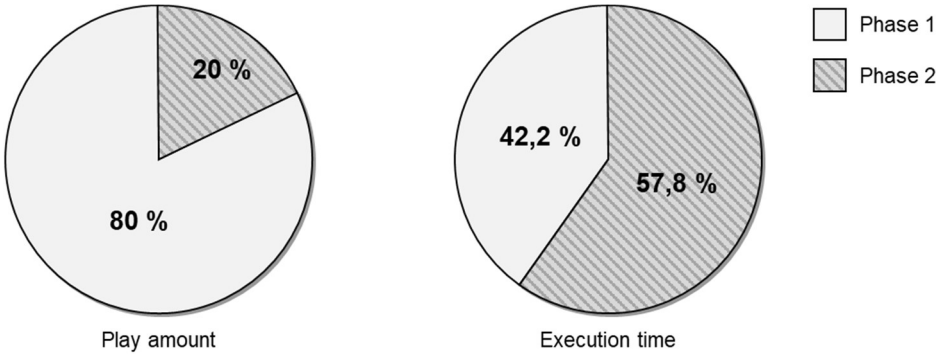


Figure 12 Comparison of number of Plays and Execution time within Top 15 Plays.

Gathered from the same play performance data, it was found that while Phase 1 setup plays made up the bulk of the complete recovery process's playlist, most of the execution time were taken by Phase 2 plays, as visualized in Figure 12.

Finally, the outputs given by Ansible during each test round showed no deviations, with each round outputting an identical execution report in comparison to each other. In addition, no major errors were experienced at any step of the test execution, were it regarding setup, disaster simulation, recovery, or cleanup effort.

### 5.4.4 Post-execution billing data

To assess the cost effects of the disaster recovery method used, an analysis on the billing data provided by AWS was done after all development and testing activities had been completed. This was done to make sure no unexpected costs were accumulating due to the usage of these cloud services by the disaster recovery tool, which would contradict the cost efficiency assumption made in this thesis. Cloudflare being a free-tier service, no such analysis was required.

According to the AWS generated billing report shown in Figure 13, total cost for all testing activities done during April, including both the primary and fallback (which included also the test site) sites was 8.09 $ US which at the current rate translates to about 7,5 €. This price includes all 10+1 test execution rounds and several developmental executions that could not be redacted from the final expenditure report.

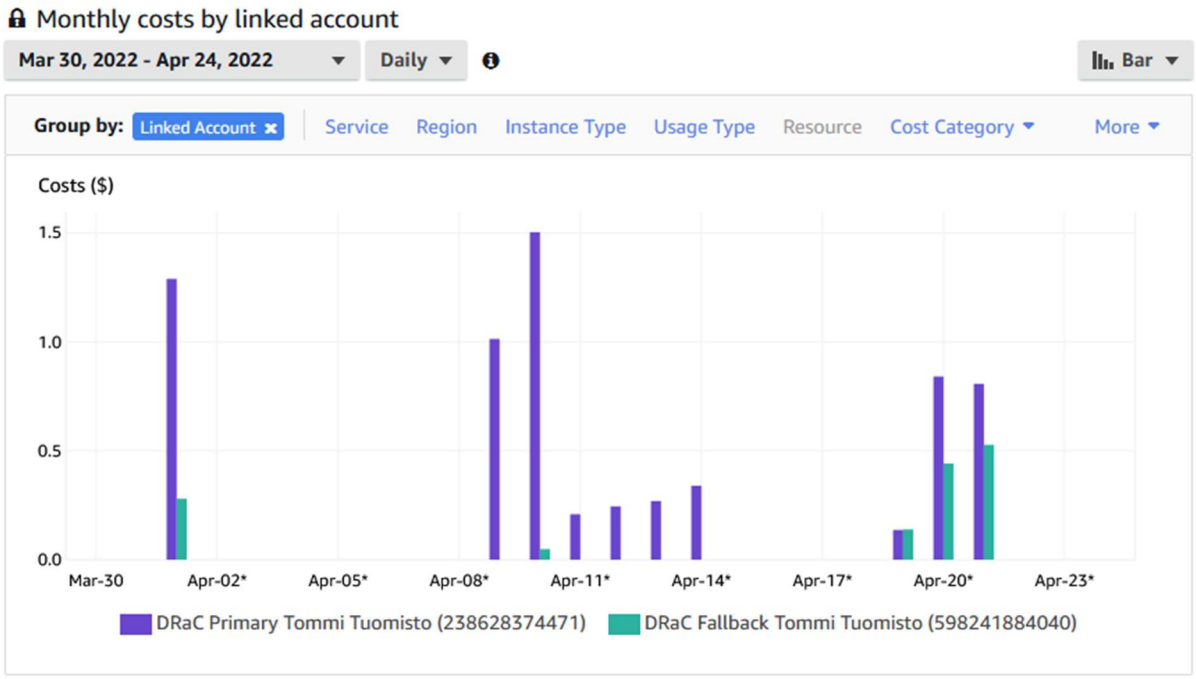


Figure 13 Accumulated AWS bill report from the testing period.

While the costs were expected to accumulate from all areas of the AWS service platform usage, only compute capacity, i.e., EC2 instance usage costs were accumulated. This is due to pricing model of AWS, which provides free capacity up to a certain threshold, after which costs start to accumulate. For example, as shown in Figure 14, during April a total of 40 GB were transferred to the EC2 instances of the fallback site and 5 GB were transferred outward, yet none of this data traffic was added to the bill.

| Bandwidth | | $0.00 |
|---|---|---|
| USD 0.0 per GB for DataTransfer-In-Bytes in EU (Stockholm) | 40.005 GB | $0.00 |
| USD 0.0 per GB for DataTransfer-Out-Bytes in EU (Stockholm) | 5.726 GB | $0.00 |
| USD 0.01 per GB for DataTransfer-Regional-Bytes in EU (Stockholm) | 0.000140 GB | $0.00 |

Figure 14 Breakdown of network transfer costs related to AWS bill.

# 6 Discussion

## 6.1 On RTA performance

As the amount of data in the tested web application portal was kept at a minimum, the achieved average 15-minute performance measure can be considered a baseline, on-top of which additional recovery time can be expected to accumulate based on the amount of stateful data the web application holds. Even still, given the allowed 8-hour limit set by relevant literature [3] for the recovery time of mission-critical systems, the achieved 15-minute average RTA performance of the tool can be said to meet and exceed the industry expectation set for legacy disaster recovery efforts.

While the effect of TTL modification to the overall recovery time performance was not explored, the average difference of 3 minutes and 24 seconds between the achieved RTA, and the total execution time of the tool suggest that lowering the TTL value from the default 5 minutes could have had a beneficial effect on overall recovery time.

## 6.2 Limiting or monitoring third-party dependencies

The problem faced with TLS certificate handling suggest that an overarching design goal for an IaC -based disaster recovery tool would be to decrease the tools reliance on third party services during the execution of the automated recovery. For example, within the created implementation, the installation of Elasticsearch depended on an external website, from where a GPG key for repository validation purposes was downloaded. Having this single third-party website non-operational, would’ve in effect halted the whole recovery process, further increasing the web application outage.

In the case of handling certificates, the original design of relying on the external service of Let’s Encrypt during the recovery process, was exchanged to copying the certificates as static content from the local development environment to the web server. In case utilizing these third-party services cannot be avoided, a suggested workaround could be for example to implement active monitoring towards these service interfaces, allowing for the possibility of reactive, correctional measures within the disaster recovery codebase, before the effect of having these services non-operational in a disastrous situation actualizes.

## 6.3 Guidelines for backups and data handling

While not covered in detail within the scope of this thesis, a key requirement for a successful recovery effort was identified to be a robust backup technique, which is kept separate from the targeted web application, and which can be set at periodic intervals to fetch the required data from the operational web application infrastructure and make it available for the disaster recovery tool to utilize. As the bulk of the 15-minutes of recovery time was observed to be spent on data handling, i.e., on copying the data from the local environment to the remote EC2 instances, and on the archiving or unarchiving processes, attention should be put on creating a fast and reliable connection between the backup data storage solution and the recovery site. Further, to limit the amount of data transferred between different components, utilizing efficient compression techniques during data handling is encouraged.

## 6.4 Disaster recovery as a service instead of IaC

With today's growing cloud services offering, it is now also possible to further abstract the DR mechanism by omitting the approach of having a tailor-made DR tool, and rather use a third-party provided disaster recovery service. For example, AWS provides the "AWS Elastic Disaster Recovery" service (AWS DRS), while Azure has its "Site Recovery" service, both of which could potentially be used for recovering the primary site of the selected web application architecture.

The total price of using AWS DRS is dominated by data replication related operations, which is why based on the data gathered for this thesis, a direct cost comparison between AWS DRS and the implemented IaC based DR tool cannot be made. In terms of actual recovery operations, the cost is estimated to be the same, as the same on-demand pricing structure applies for both approaches in the event of an actual disaster. [87]

Nevertheless, even if an additional cost would be incurred, certain benefits that come with a service model cannot be neglected when considering the best approach for one's disaster recovery purposes. For one, a service is typically provided with a set of guarantees, which the service user can rely upon. Having the capability to provide similar guarantees with a self-made tool could potentially add unexpected cost in testing the implemented solution. Second, the work for implementing a tailor-made tool can be considered heavier than with a service. For example, the AWS DRS service advertises that it can be taken to use "without specialized skillsets" [88].

Given the IaC based design proposed by this thesis, the downside of using a service on the other hand is considered two-fold:

1) The design proposed in this thesis is beneficial in situations where the infrastructure is already being managed through IaC definitions. With the proposed two-phase split, infrastructure management teams could potentially leverage their existing definitions for disaster recovery by only implementing the 2nd phase data recovery steps. By utilizing a DR recovery service, the management of DR would most likely be a separately managed mechanism.

2) Resorting to use another cloud service for this purpose would mean that a similar level of service could be offered by anyone, with transparent pricing structure, removing the ability to differentiate from others operating in the same market environment.

# 7 Conclusion

For designing an IaC -based disaster recovery tool, this thesis proposed a primary-fallback oriented disaster recovery model, where the fallback site of the infrastructure is a pre-created, yet completely empty, or 'cold', cloud service account, into which a near duplicate copy of the primary site is recreated in the event of a disaster. Within this model, a two phased procedure for recovery is proposed, where the 2nd phase stateful application data recovery procedure is kept as an add-on functionality to the 1st phase stateless infrastructure management practices. By operating both phases within the same IaC -project structure, the proposed model allows for both phases to operate on a single source of truth, thus ensuring that the 2nd phase recovers the application infrastructure to its most recent state. For switching from primary to fallback site, the tool was designed to utilize a DNS failover mechanism, whereby modifying the DNS A-record associations of the Public IP address during the start of the recovery process, the end-user traffic was automatically transferred to the recovered site once the TTL timer for end-user devices expire.

Based on the insights and data gathered during the testing of the proposed design, the IaC -based disaster recovery tool created with Ansible and Terraform can be considered functional and performant within the known limits set by industry established practices. Further, based on the accumulated costs observed after all testing activities had completed, having an IaC -based fallback mechanism was observed to offer a near cost-free redundancy mechanism, as no expenditure for upkeeping the infrastructure definitions or the cold fallback site were identified while the primary site was operational, and no significant costs that could be linked to the actual recovery process were observed.

Because of these two findings, overall result is that using IaC -based disaster recovery mechanisms as a part of one's service offering, even with web applications of less criticality, is an avenue worth increased consideration.

However, it should be highlighted that the findings presented in this thesis are not generalizable outside of the spectrum of the selected tools and cloud services used for the implementation and testing of the selected design. While the findings of this thesis propose Terraform and Ansible as a working combination for the orchestration and disaster recovery of specific AWS resources, the same cannot be assumed when using these tools against another cloud service platform and its resources. Similarly, as there is a wide range of IaC

orchestration tools to select from, having one set of tools proven functional doesn't guarantee a similar outcome with another, different set of tools.

Given this shortcoming, for future research the scope of disaster recovery testing effort could be widened to cover a larger array of different cloud service platforms, against which the design could be tested against. Alternatively, to achieve even higher levels of redundancy and geo-local separation, a multi-cloud disaster recovery system, where the primary and fallback sites are in separate cloud service platforms, could be an interesting angle to explore.

## References


[1] Statistics Finland, "Tietotekniikan käyttö yrityksissä: 3. Pilvipalvelut," 2021. [Online]. Available: https://tilastokeskus.fi/til/icte/2021/icte_2021_2021-12-03_kat_003_fi.html.

[2] DORA, Google Cloud, "The Accelerate State of DevOps Report," 2019. [Online]. Available: https://services.google.com/fh/files/misc/state-of-devops-2019.pdf

[3] K. Schmidt. High Availability and Disaster Recovery: Concepts, Design, Implementation. Berlin, Germany: Springer, 2006.

[4] J. R. Vacca. Cyber Security and IT Infrastructure Protection. Waltham, United States: Syngress, 2014.

[5] European Banking Authority, "EBA Guidelines on ICT and security risk management," 2019. [Online]. Available: https://www.eba.europa.eu/regulation-and-policy/internal-governance/guidelines-on-ict-and-security-risk-management

[6] J. Singh, "Understanding RPO and RTO," 2021. [Online]. Available: https://www.druva.com/blog/understanding-rpo-and-rto/.

[7] FIN-FSA, "Management of operational risk in supervised entities of the financial sector," 2014. [Online]. Available: https://www.finanssivalvonta.fi/globalassets/en/regulation/fin-fsa-regulations-and-guidelines/2014/08_2014/08_2014.m4_en.pdf.

[8] FIN-FSA, "Legal framework of FIN-FSA regulations and guidelines," 2018. [Online]. Available: https://www.finanssivalvonta.fi/en/regulation/legal-framework/.

[9] FINLEX, "Laki sähköisen viestinnän palveluista 917/2014," 2022. [Online]. Available: https://www.finlex.fi/fi/laki/smur/2014/20140917

[10] OTAVA, "Disaster Recovery," 2019. [Online]. Available: https://web.otava.com/disaster-recovery-answers-all-in-one-place

[11] P. H. Gregory, IT disaster recovery planning for dummies. Hoboken, United States: Wiley, 2007.

[12] J.W. Rittinghouse, J.F. Ransome. Business Continuity and Disaster Recovery for Infosec Managers. Amsterdam, United States: Elsevier Digital Press, 2005.

[13] Ministry of Finance, "ICT -varautumisen vaatimukset," 2012. [Online]. Available: https://vm.fi/documents/10623/307669/ICT-varautumisen+vaatimukset/9fa21bee-efcc-485a-8677-4eb4e0a2fa1f/ICT-varautumisen+vaatimukset.pdf

[14] R: Atkinson, "Communicating and Staffing for Unplanned Outages," 2012. [Online]. Available: https://www.thinkhdi.com/~/media/HDICorp/Files/White-Papers/whtppr-1212-atkinson-outages

[15] Red Hat, "What is IT infrastructure?," 2019. [Online]. Available: https://www.redhat.com/en/topics/cloud-computing/what-is-it-infrastructure.

[16] Y. Liu, Y. Yue, L. Guo. UNIX Operating System: The Development Tutorial via UNIX Kernel Services. Beijing, China: Springer, 2011.

[17] IBM, "What is IT Infrastructure?". [Online]. Available: https://www.ibm.com/topics/infrastructure. (Accessed 20.2.2022)

[18] R. Winkelman, "What is Networking Hardware?." [Online]. Available: https://fcit.usf.edu/network/chap3/chap3.htm. (Accessed 20.2.2022)

[19] J. Hoffman, "Modern Infrastructure: The Convergence of Network, Compute, and Data," 2013. [Online]. Available: https://www.usenix.org/conference/lisa13/technical-sessions/keynote/hoffman.

[20] Wiley, "Components of a Server Computer," 2016. [Online]. Available: https://www.dummies.com/article/technology/information-technology/networking/general-networking/components-of-a-server-computer-200671/.

[21] IBM, "What is data storage?". [Online]. Available: https://www.ibm.com/topics/data-storage. (Accessed 20.2.2022)

[22] F. Douglis, O. Krieger. “Virtualization”, IEEE internet computing, Vol. 17, pp. 6–9, 2013.

[23] M. Pearce, S. Zeadally, R. Hunt. "Virtualization: Issues, Security Threats, and Solutions", ACM Computing Surveys, Vol. 45, No. 2, Article 17, 2013.

[24] J. Mickens, "CS 161: Lecture 16 Virtualization," 2017. [Online]. Available: https://www.eecs.harvard.edu/~cs161/notes/virtualization.pdf

[25] D. Drutskoy, E. Keller, J. Rexford, “Scalable Network Virtualization in Software Defined Networks”, IEEE internet computing, Vol. 17, no. 2, pp.20-27, 2013.

[26] VMware, "What is network virtualization?". [Online]. Available: https://www.vmware.com/topics/glossary/content/network-virtualization.html. (Accessed 16.3.2022)

[27] Juniper, "What is VXLAN?". [Online]. Available: https://www.juniper.net/us/en/research-topics/what-is-vxlan.html. (Accessed 16.3.2022)

[28] DataCore, "Storage Virtualization". [Online]. Available: https://www.datacore.com/storage-virtualization/. (Accessed 16.3.2022)

[29] D. Bermach, E. Wittern, S. Tai. Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective. Springer, 2017.

[30] W3Techs, "Usage statistics of Unix for websites". [Online]. Available: https://w3techs.com/technologies/details/os-unix. (Accessed 20.3.2022)

[31] AWS, "Amazon EC2 security groups for Linux instances". [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-security-groups.html. (Accessed 20.3.2022)

[32] Microsoft, "Network security groups," 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-network/network-security-groups-overview.

[33] Stackpath, "What is infrastructure as code?". [Online]. Available: https://www.stackpath.com/edge-academy/what-is-infrastructure-as-code. (Accessed 24.3.2022)

[34] AWS, "Infrastructure as Code". [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/infrastructure-as-code.html. (Accessed 24.3.2022)

[35] J. Klein, D. Reynolds. "Infrastructure as Code-Final Report," 2018. [Online]. Available: https://resources.sei.cmu.edu/asset_files/WhitePaper/2019_019_001_539335.pdf.

[36] Pulumi, "Delivering Cloud Native Infrastructure as Code". [Online]. Available: https://cdn2.hubspot.net/hubfs/4429525/Content/Pulumi-Delivering-CNI-as-Code.pdf.

[37] W. Hummer, F. Rosenberg, F. Oliveira, T. "Testing Idempotence for Infrastructure as Code," 2013. [Online]. Available: https://www.researchgate.net/publication/255978303_Testing_Idempotence_for_Infrastructure_as_Code

[38] N. Singh Gill, "Infrastructure as Code Tools to Boost Your Productivity in 2022," 2022. [Online]. Available: https://www.nexastack.com/blog/best-iac-tools.

[39] A. Valdes, "The best Infrastructure as Code tools for 2022," 2022. [Online]. Available: https://www.clickittech.com/devops/infrastructure-as-code-tools/.

[40] F. Pialoux, "Best Infrastructure as Code Tools (IaC): The Top 10 for 2022," 2022. [Online]. Available: https://bluelight.co/blog/best-infrastructure-as-code-tools.

[41] B. O'Sullivan, "Mercurial: The Definitive Guide," 2009. [Online]. Available: http://hgbook.red-bean.com/read/how-did-we-get-here.html.

[42] K. Geisshirt, E. Zattin, A. Olsson, R. Voss. Git Version Control Cookbook, 2nd edition. Birmingham, UK: Packt, 2018.

[43] Synopsys, "Compare Repositories". [Online]. Available: https://www.openhub.net/repositories/compare. (Accessed 10.2.2022)

[44] L. Shklar, R. Rosen. Web Application Architecture: Principles, Protocols and Practices, 2nd Edition. Chichester, UK: Wiley, 2009.

[45] Liferay, "Our story: Where we began". [Online]. Available: https://www.liferay.com/company/our-story. (Accessed 10.2.2022)

[46] Discussions with J. Haaja, Ambientia, 20.9.2021, 25.10.2021.

[47] A. Availa, "Datakatsaus: Valtionhallinnon julkaisujärjestelmät Suomessa," 2020. [Online]. Available: https://web-ostajanopas.fi/2020/08/25/datakatsaus-valtionhallinnon-julkaisujarjestelmat-suomessa/.

[48] Ch5 Finland, "Liferay Portal CE ja EE ovat kaksi eri tuotetta," 2017. [Online]. Available: https://www.planeetta.fi/2017/07/19/liferay-portal-ce-ja-ee-ovat-kaksi-eri-tuotetta/.

[49] IBM Cloud Education, "Three-Tier Architecture," 2020. [Online]. Available: https://www.ibm.com/cloud/learn/three-tier-architecture.

[50] Liferay, "Liferay DXP 7.2 Compatibility Matrix," 2021. [Online]. Available: https://help.liferay.com/hc/en-us/articles/360028982631-Liferay-DXP-7-2-Compatibility-Matrix.

[51] Liferay, "Elasticsearch". [Online]. Available: https://learn.liferay.com/dxp/latest/en/using-search/installing-and-upgrading-a-search-engine/elasticsearch.html.

[52] Liferay, "Database Configuration". [Online]. Available: https://learn.liferay.com/dxp/latest/en/installation-and-upgrades/reference/database-configurations.html.

[53] Liferay, "Understanding Search and Indexing". [Online]. Available: https://help.liferay.com/hc/en-us/articles/360034199432-Understanding-Search-and-Indexing.

[54] Liferay, "Deploy Liferay DXP in Azure," 22. [Online]. Available: https://www.liferay.com/resources/whitepapers/Deploying+Liferay+DXP+in+Azure.

[55] Liferay, "Deploy Liferay DXP in Amazon Web Services," 2022. [Online]. Available: https://www.liferay.com/resources/whitepapers/Deploying+Liferay+DXP+in+Amazon+Web+Services.

[56] AWS, "Cloud computing with AWS". [Online]. Available: https://aws.amazon.com/what-is-aws/. (Accessed 24.03.2022)

[57] AWS, "What is Amazon VPC?". [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html.

[58] AWS, "Security group rules". [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/security-group-rules.html.

[59] AWS, "Amazon EC2 Instance Types". [Online]. Available: https://aws.amazon.com/ec2/instance-types/.

[60] AWS, "Connect to the internet using an internet gateway". [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html.

[61] AWS, "Amazon EBS volumes". [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volumes.html.

[62] AWS, "Amazon Machine Images (AMI)". [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html.

[63] Cloudflare, "Infrastructure as Code". [Online]. Available: https://www.cloudflare.com/partners/infrastructure-as-code/.

[64] Cloudflare, "So what is Cloudflare?". [Online]. Available: https://www.cloudflare.com/learning/what-is-cloudflare/.

[65] NGINX, "What is a Reverse Proxy vs. Load Balancer?". [Online]. Available: https://www.nginx.com/resources/glossary/reverse-proxy-vs-load-balancer/.

[66] SSL.com, "What Is a Certificate Authority (CA)?," 2021. [Online]. Available: https://www.ssl.com/faqs/what-is-a-certificate-authority/.

[67] AWS, "AWS Certificate Manager". [Online]. Available: https://aws.amazon.com/certificate-manager/.

[68] Let's Encrypt, "About Let's Encrypt". [Online]. Available: https://letsencrypt.org/about/.

[69] Let's Encrypt, "How It Works". [Online]. Available: https://letsencrypt.org/how-it-works/.

[70] Electronic Frontier Foundation, "about certbot". [Online]. Available: https://certbot.eff.org/pages/about.

[71] AWS, "AWS Certificate Manager Pricing". [Online]. Available: https://aws.amazon.com/certificate-manager/pricing/.

[72] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "RFC2616 - Hypertext Transfer Protocol -- HTTP/1.1," 1999. [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec15.html.

[73] Google, "HTTPS encryption on the web". [Online]. Available: https://transparencyreport.google.com/https/overview.

[74] F. Gont, "Introduction to DNS Privacy," 2019. [Online]. Available: https://www.internetsociety.org/resources/deploy360/dns-privacy/intro/.

[75] Mozilla Contributors: https://developer.mozilla.org/en-US/docs/MDN/About/contributors.txt, "What is a URL?," 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_URL.

[76] Cloudflare, "DNS A record". [Online]. Available: https://www.cloudflare.com/learning/dns/dns-records/dns-a-record/.

[77] Cloudflare, "Cloudflare DNS FAQ". [Online]. Available: https://support.cloudflare.com/hc/en-us/articles/360017421192-Cloudflare-DNS-FAQ.

[78] Hashicorp, "What is Terraform?". [Online]. Available: https://www.terraform.io/intro.

[79] Hashicorp, "The Core Terraform Workflow". [Online]. Available: https://www.terraform.io/intro/core-workflow.

[80] Hashicorp, "Purpose of Terraform State". [Online]. Available: https://www.terraform.io/language/state/purpose.

[81] Hashicorp, "Workspaces". [Online]. Available: https://www.terraform.io/language/state/workspaces.

[82] Red Hat, "Overview: How Ansible Works". [Online]. Available: https://www.ansible.com/overview/how-ansible-works.

[83] Let's Encrypt, "Rate Limits". [Online]. Available: https://letsencrypt.org/docs/rate-limits/.

[84] Liferay, "Backing Up". [Online]. Available: https://learn.liferay.com/dxp/latest/en/installation-and-upgrades/maintaining-a-liferay-installation/backing-up.html.

[85] Marcus J. Ranum, “Thinking About Firewalls,” Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II), 1992.

[86] SSH, "SSH Command - Usage, Options, Configuration". [Online]. Available: https://www.ssh.com/academy/ssh/command.

[87] AWS, "AWS Elastic Disaster Recovery pricing". [Online]. Available: https://aws.amazon.com/disaster-recovery/pricing/.

[88] AWS, “AWS Elastic Disaster Recovery”. [Online]. Available: https://aws.amazon.com/disaster-recovery/.

# Appendices

## Appendix 1 Disaster recovery tool directory structure and content

```
.
├── backups
│   ├── db
│   │   └── dump.sql
│   └── lr
│       ├── data.tar.gz
│       ├── license.tar.gz
│       ├── osgi.tar.gz
│       └── properties.tar.gz
├── group_vars
│   └── ec2.yml
├── roles
│   ├── p11_setup_aws_cf
│   │   └── tasks
│   │       └── main.yml
│   ├── p12_setup_web
│   │   ├── files
│   │   │   ├── any.liferayapp.xyz-cert.pem
│   │   │   ├── any.liferayapp.xyz-chain.pem
│   │   │   ├── any.liferayapp.xyz-fullchain.pem
│   │   │   ├── any.liferayapp.xyz-privkey.pem
│   │   │   └── tls_certificate_placeholder
│   │   ├── tasks
│   │   │   └── main.yml
│   │   └── templates
│   │       └── nginx.conf.j2
│   ├── p13_setup_app
│   │   ├── files
│   │   │   └── liferay-portal.service
│   │   └── tasks
│   │       └── main.yml
│   ├── p14_setup_db
│   │   ├── files
│   │   │   ├── pg_hba.conf
│   │   │   ├── postgresql.conf
│   │   │   └── postgres.repo
│   │   └── tasks
│   │       └── main.yml
│   ├── p15_setup_search
│   │   ├── files
│   │   │   ├── elasticsearch.repo
│   │   │   └── elasticsearch.yml
│   │   └── tasks
│   │       └── main.yml
│   ├── p16_integrate
│   │   ├── files
│   │   │   └── osgi-es.config
│   │   ├── tasks
│   │   │   └── main.yml
│   │   └── templates
│   │       ├── portal-setup-wizard.properties.j2
│   │       └── server.xml.j2
│   ├── p21_stop_web
│   │   └── tasks
```

```
│   │   │       └── main.yml
│   │   ├── p22_stop_app
│   │   │   └── tasks
│   │   │       └── main.yml
│   │   ├── p23_restore_db
│   │   │   └── tasks
│   │   │       └── main.yml
│   │   ├── p24_restore_app
│   │   │   └── tasks
│   │   │       └── main.yml
│   │   ├── p25_start_app
│   │   │   └── tasks
│   │   │       └── main.yml
│   │   └── p26_start_web
│   │       └── tasks
│   │           └── main.yml
│   ├── software
│   │   └── liferay-ce-portal-tomcat-7.4.2-ga3-20210728053338694.tar.gz
│   ├── terraform
│   │   ├── shell
│   │   │   └── init_ssh.sh
│   │   ├── terraform.tfstate.d
│   │   │   ├── fallback
│   │   │   │   └── terraform.tfstate
│   │   │   ├── primary
│   │   │   │   └── terraform.tfstate
│   │   │   └── test
│   │   │       └── terraform.tfstate
│   │   ├── tfvars
│   │   │   ├── fallback.tfvars
│   │   │   ├── primary.tfvars
│   │   │   └── test.tfvars
│   │   ├── dns.tf
│   │   ├── ec2.tf
│   │   ├── main.tf
│   │   ├── network.tf
│   │   ├── output.tf
│   │   ├── security.tf
│   │   └── variables.tf
│   ├── disaster_recovery.yml
│   ├── inventory
│   └── run_backup.sh
```

## Appendix 2 Version control ".gitignore"

```
terraform/.terraform
terraform/.terraform.lock.hcl
terraform/*.tfstate
terraform/tfvars*
terraform/terraform.tfstate.d/primary/*.tfstate*
terraform/terraform.tfstate.d/fallback/*.tfstate*
terraform/terraform.tfstate.d/test/*.tfstate*
software/liferay-ce-portal-tomcat-*
roles/p12_setup_web/files/*.pem
backups/db/*sql
backups/lr/*gz
```

## Appendix 3 Example resulting output of an Ansible recovery execution

```
PLAY RECAP
*********************************************************************
18.158.203.154:  ok=11  changed=9   unreachable=0  failed=0  skipped=0
  rescued=0  ignored=0
3.127.67.219:    ok=14  changed=13  unreachable=0  failed=0  skipped=0
  rescued=0  ignored=0
3.73.81.225:     ok=9  changed=7    unreachable=0  failed=0  skipped=0
  rescued=0  ignored=0
52.28.103.80:    ok=25 changed=21   unreachable=0  failed=0  skipped=0
  rescued=0  ignored=0
localhost:      ok=8   changed=5    unreachable=0  failed=0  skipped=1
  rescued=0  ignored=0

Thursday 21 April 2022  20:26:58 +0300 (0:00:02.396)       0:11:48.168 **
=====================================================================
p24_restore_app : Copy Liferay OSGi library from backup --------- 202.82s
p13_setup_app : Copy Liferay installation file ------------------ 118.93s
p11_setup_aws_cf : Run Terraform apply ---------------------------- 44.87s
p24_restore_app : Restore OSGi ------------------------------------ 38.36s
p13_setup_app : Unarchive the installation file ------------------ 22.66s
p15_setup_search : Start the Elasticsearch service --------------- 21.88s
p15_setup_search : Install the latest version of Elasticsearch.. - 19.52s
p13_setup_app : Install OpenJDK ----------------------------------- 19.32s
Wait 600 seconds for target connection to become reachable.. ----- 18.44s
p14_setup_db : Install the latest version of PostgreSQL13 -------- 18.35s
p14_setup_db : Install the latest version of PostgreSQL13 -------- 17.34s
p23_restore_db : Restore from dump file --------------------------- 9.49s
p15_setup_search : Install required analysis-kuromoji plugin ------ 8.41s
p15_setup_search : Install required analysis-smartcn plugin ------- 8.35s
p14_setup_db : Install EPEL package library ----------------------- 8.28s
p12_setup_web : Install NGINX ------------------------------------- 8.21s
p15_setup_search : Install required analysis-stempel plugin ------- 8.11s
p15_setup_search : Install required analysis-icu plugin ----------- 8.02s
p23_restore_db : Copy database dump file from backup -------------- 4.19s
p12_setup_web : Copy TLS certificate private key ------------------ 3.89s
```