Universidad Nacional de La Plata

Facultad de Informática

Infraestructura como Código
Caso de estudio: Cientópolis

Tesis presentada por Matias Damian Banchoff Tzancoff
Dirigida por Alejandro Fernandez y Matias Urbieta
para obtener el grado de Maestría en Ingeniería de Software

2019

Agradecimientos

Las herramientas usadas en este trabajo (a excepción de AWS) y las usadas para escribir este documento son *open source*, por lo que en primer lugar me gustaría agradecer a los desarrolladores que han liberado tantas aplicaciones bajo una licencia *open source*.

Me gustaría agradecer a Alejandro Fernandez y Diego Torres por haberme guiado y dedicado su tiempo y atención en el transcurso de esta tesis. A Juan Pablo Lozano por su colaboración técnica en lo relacionado con el ambiente de Proxmox. Sin su participación este trabajo no habría sido posible.

También quiero agradecer enormemente a mi familia, quienes me han dado el tiempo y el espacio para llevar a cabo esta tesis. Han sabido apoyarme y motivarme en los momentos necesarios. Y también han sabido distraerme en los momentos justos.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

> Nos quitaron el tiempo y nos dieron el reloj
>
> *Abdullah Ibrahim*

En la última década la complejidad de los sistemas informáticos se ha incrementado notablemente, siendo más costoso en términos de esfuerzo el instanciar y configurar una plataforma para ejecutar estos sistemas. Además, la velocidad con la que se debe entregar software al mercado es cada vez más demandante, principalmente en el ámbito de las aplicaciones web, lo cual genera una mayor presión sobre los desarrolladores y los administradores (Feitelson, D., Frachtenberg, E. & Beck, K., 2013). Las metodologías ágiles atacan esta problemática desde el lado del desarrollo, mientras que la arquitectura programada lo hace desde la perspectiva de la administración de servidores.

Como se menciona en el párrafo anterior, los sistemas informáticos tienden a ser grandes sistemas distribuidos, compuestos de partes heterogéneas y dependientes de una variedad de otros servicios. Todo esto implica que la instalación, configuración y mantenimiento de los sistemas informáticos se ha complejizado. Más aún si se considera que las metodologías ágiles requieren de varios ambientes donde ejecutar las aplicaciones (desarrollo, testing, staging, etc), lo cual termina duplicando, al menos, el esfuerzo de los administradores (Cois, C., Yankel, J. & Connell, A., 2015).

La respuesta que ha dado la Informática a estos problemas es la sistematización y automatización del despliegue de aplicaciones y de la configuración de los servidores. Actualmente es posible configurar servidores programáticamente y sin intervención de los administradores. Esto es de especial importancia cuando se trata de entornos donde no se dispone de personal técnico capacitado. A este tipo de prácticas, que consideran a la infraestructura como un artefacto más a ser gestionado de manera análoga al software, se las conoce como **infraestructura programable** o **infraestructura como código**[1](Huttermann, M., 2012).

La arquitectura tiene por finalidad satisfacer los requerimientos dados por los ***stakeholders***[2] en un grado tal acordado con ellos. Es importante que la arquitectura conserve dichas propiedades[3]. Esta cuestión de conservar las propiedades es difícil de satisfacer cuando el proyecto no dispone de personal experto en la gestión de infraestructura (servidores, nubes, etc.), por lo que una "fácil instanciación" pasa a ser otro requisito de la arquitectura. Es esencial que la arquitectura esté documentada correctamente para que se pueda continuar con su desarrollo y mantenimiento y que se la pueda transferir sin problemas a otros grupos de investigación o innovación tecnológica.

Existen, también, metodologías formales para evaluar arquitecturas de software con el objetivo de asegurar un determinado grado de calidad aceptable para los *stakeholders*. Al evaluar la arquitectura se obtiene un mejor entendimiento de los ***trade-offs***[4] involucrados en el diseño de un sistema, que determinan la calidad de la arquitectura y del sistema. Se puede utilizar diferentes metodologías[5] para evaluar una arquitectura según el grado de cumplimiento con determinadas propiedades.

[1]El concepto de infraestructura como código hace referencia a la posibilidad de configurar y administrar *datacenters* enteros usando archivos de configuración ejecutables por un programa. De esta manera se puede versionar y resguardar las diferentes configuraciones que puede tomar un *datacenter*.

[2]El término *stakeholder* se puede traducir como **parte interesada** y hace referencia a las personas u organismos que se ven afectados por las decisiones del proyecto. En este trabajo se utilizará el término original en inglés.

[3]Como un determinado grado de performance o un nivel de seguridad bien definido.

[4]Se entiende por *trade-off* a una **solución de compromiso** en donde se degrada o pierde una cualidad en favor de otra. Por ejemplo, se degrada el desempeño para ganar seguridad mediante el cifrado de los datos. En este trabajo se prefiere el término en inglés.

[5]En la actualidad existen diversos métodos de evaluación de arquitecutras de sistemas, siendo ATAM uno de los más antiguos y conocidos.

Estas metodologías se basan en priorizar determinadas cualidades deseables en la arquitectura con el fin de asegurar que responde a los requerimientos planteados. Uno de los métodos existentes para la evaluación de arquitecturas es LAAAM[6], caracterizado por ser un método *lightweight* y *agile*, en consonancia con las metodologías de desarrollo de software modernas.

1.1. Objetivos del presente trabajo

El objetivo general de esta tesis es evaluar, en un caso de estudio concreto, la aplicabilidad e impacto de los principios de infraestructuras como código en la calidad de las arquitecturas de proyectos de investigación e innovación tecnológica, y en los procesos de gestión de las mismas. Como objetivos específicos se tienen:

- En un proyecto concreto de investigación e innovación tecnológica (Cientópolis), estudiar las necesidades de arquitectura y procesos de gestión de las mismas, relevar las prácticas utilizadas en la actualidad, e identificar problemáticas y falencias.
- Proponer e implementar una arquitectura nueva para el proyecto y procesos de gestión de la misma, en el marco de las prácticas de infraestructuras programables o infraestructuras como código.
- Evaluar la arquitectura y procesos de gestión propuestos, y analizar el impacto en relación a la situación original del proyecto bajo estudio.

1.2. Enfoque general

A grandes rasgos la metodología a seguir se basa en los siguientes pasos:

- Elegir una metodología de evaluación de arquitecturas.
- Recopilar información sobre la arquitectura actual y exigencias para la nueva utilizando encuestas para los *stakeholders*.

[6]Lightweight Architecture Alternative Assessment Method, derivado de ATAM

- Analizar y documentar las exigencias planteadas para la nueva arquitectura.
- Utilizar una de las metodologías de evaluación de arquitecturas para desarrollar una propuesta que responda a las exigencias del proyecto Cientópolis.
- Implementar la arquitectura planteada utilizando para su gestión herramientas de automatización y evaluar los resultados.

En el proceso se verán los siguientes temas de investigación:

Metodologías de evaluación de arquitecturas: mecanismos utilizados por los administradores y demás *stakeholders* para acordar un determinado nivel de calidad por parte de la arquitectura implementada.

Arquitectura como código: describir y administrar la arquitectura como si fuera el código de un programa, permitiendo versionarla, automatizar la configuración e instanciar diferentes ambientes con facilidad.

***Devops* y herramientas asociadas:** El concepto de *devops* se refiere a la interdependencia entre los desarrolladores y los administradores. Es un perfil intermedio. Existen varias herramientas asociadas a este nuevo campo, como Ansible, Terraform o Docker, entre muchas otras.

1.3. Nociones básicas de arquitectura

La mayoría de los autores dan su propia definición de "arquitectura". Por ejemplo, M. Javed en su tesis (Javed, M., 2007) menciona que Lex Bijlsma la define como la estructura global del sistema, sus componentes y las relaciones entre esos componentes. Los componentes tienen una interface y son los *building blocks* de la arquitectura. La ISO 42010 (ISO 42010, 2011) amplía el concepto indicando que se deben considerar el contexto y las decisiones de diseño como parte de la arquitectura.

Rick Kazman da una definición similar en su libro Software Architecture in Practice(Bass, L., Clements, P. & Kazman, R., 2012), agregando a la definición de "arquitectura" las propiedades visibles externamente de las componentes de las que habla Bijlsma. Estas propiedades visibles externamente son suposiciones que pueden hacer otras componentes acerca de una componente. Es decir, la interface y el comportamiento exhibido por una componente, pero no su implementación. Por otra parte, M. Fowler menciona en su artículo ¿Quién necesita un arquitecto? (Fowler, M., 2003) que Ralph Johnson define el término "arquitectura" como **todo lo que es importante**[7]

Además, en otros escritos se habla de la necesidad de una "buena" arquitectura para el éxito de un proyecto de software. Pero más allá de la definición que se dé o adopte, sí está claro que la arquitectura de un sistema ayuda a satisfacer las inquietudes (*concerns*, en la bibliografía en inglés) de los *stakeholders*.

Todo arquitecto de software dirá que cualquier sistema debe cumplir con varios requisitos, como ser seguro, mantenible, tener un buen desempeño (*performance*), etc. La cuestión es que a menudo estos requisitos se deben negociar. Por ejemplo: agregar encriptación a un sistema lo hace más seguro, pero baja su desempeño. Luego, la tarea del arquitecto es identificar todos estos requisitos dados por los *stakeholders* y proponer una arquitectura que los satisfaga en la medida justa o necesaria.

Del párrafo anterior se deduce una cuestión interesante: la primera preocupación del arquitecto no es la funcionalidad del sistema, sino manejar las inquietudes de los *stakeholders*. En particular, es la satisfacción de los atributos relacionados con la calidad del sistema la principal preocupación del arquitecto. Pero, también se explicó que no es posible satisfacer completamente todo requerimiento. Entonces, ¿cómo logra el arquitecto alcanzar el grado de satisfacción correcto para cada requisito? Esa respuesta la dan las metodologías de evaluación de arquitecturas.

[7] En inglés, *the important stuff (whatever that is)*.

En lo que respecta a este trabajo, la definición de Bijlsma dada anteriormente para la arquitectura de un sistema (y al hablar de "sistema" se abarca tanto *software* como *hardware*) junto con los agregados de la ISO 42010, que sirve para describir y documentar arquitecturas de sistemas (se verá con más detalle en el Capítulo-5) funcionan bien y es lo que el lector deberá tener en mente cuando se haga referencia a una arquitectura.

1.4. Infraestructura vs. arquitectura

En algunas ocasiones los términos "infraestructura" y "arquitectura" se utilizan como sinónimos, pero en este trabajo haremos una distinción. Se puede ver a la infraestructura como aquello que atraviesa a toda la organización, mientras que a la arquitectura, como una solución a una problemática en particular. La infraestructura brinda servicios a diferentes sistemas. Estos sistemas tienen una arquitectura particular.

Un ejemplo concreto: la arquitectura de Cientópolis se puede instanciar para varios proyectos de ciencia ciudadana, pero todas esas instanciaciones usaran los mismos servicios de la infraestructura (routers, switches, mensajería, monitoreo, etc).

Qué compete a la arquitectura y qué a la infraestructura es complicado definir y la linea divisoria muy delgada. En principio, si un servicio lo usa más de un sistema, entonces se dirá que es un servicio de la infraestructura, mientras que si sólo lo usa un sistema, entonces será un servicio propio de la arquitectura.

1.5. Problemas derivados de una arquitectura incorrecta

En esta sección se mencionan algunas características que suelen exhibir las arquitecturas (tales como seguridad, usabilidad, etc) que serán tratadas más adelante. Por el momento se espera que el lector maneje una definición intuitiva de esos conceptos, que serán explicados en mayor detalle en el siguiente capítulo.

Cuando en ésta y otras secciones se hable de "arquitectura incorrecta" el lector deberá entender que se trata de arquitecturas que no son acordes a las necesidades de las aplicaciones que la habitan, de los administradores y desarrolladores que la utilizan y/o que no se *llevan bien* con la infraestructura existente.

La baja *usabilidad* de una arquitectura puede ser un indicativo de que se está empleando una arquitectura incorrecta, que se traduce en trabajo duplicado por parte de los administradores o necesidad de realizar cambios en varios lugares para corregir un problema, por mencionar sólo dos ejemplos. También instalar y configurar las aplicaciones se torna una tarea más compleja de lo que realmente debería ser, poniendo en riesgo la fiabilidad de todo el sistema cada vez que se agrega, modifica o remueve una aplicación.

Mantener una arquitectura incorrecta es también más costoso en términos de tiempo y de esfuerzo humano. Igualmente, extender el sistema con nueva funcionalidad pasa a ser un problema mayor cuando no se trabaja con una arquitectura bien diseñada.

Por otra parte, también podría darse el caso de que la arquitectura no esté diseñada para escalar en el tiempo, de modo que la aplicación podría tener problemas a futuro si la carga o la cantidad de usuarios creciera.

Por último, también la seguridad de todo el sistema se ve comprometida. Por ejemplo, la arquitectura podría no soportar SSL/TLS para las aplicaciones.

1.6. Problemas derivados de una gestión incorrecta de la arquitectura

El principal problema derivado de una incorrecta gestión de la arquitectura es la merma en la estabilidad del sistema. Asignación de roles poco clara o directamente inexistente genera caos en el proyecto, que se manifiesta de varias maneras en los servidores, como por ejemplo:

- los tiempos de respuesta ante un problema se extienden debido a que no se tiene claro los roles, y por ende no se sabe quién debe actuar para brindar la solución.

- tarde o temprano la configuración de los servicios termina pareciendo más un *collage* que archivos de configuración productivos. Esto se debe a la inexistencia de reglas para escribir las configuraciones y cada administrador, o usuario que tenga acceso administrativo al servidor, termina aplicando su propio criterio.
- la falta de estadísticas de uso impide conocer el estado actual de los servicios y las necesidades de cara a un futuro crecimiento.
- si no se configuran herramientas de monitoreo los administradores siempre irán detrás de los problemas.

La gestión de la documentación también es de suma importancia, dado que es el mecanismo por el cual se hace la transferencia de conocimiento a nuevos integrantes del proyecto. Todas las carencias que presente la documentación se verán materializadas en problemas a la hora de entender la arquitectura, proponer y realizar cambios y enseñar su uso a nuevos miembros del grupo.

1.7. Resumen

En este capítulo se dio un significado para el término "arquitectura" de un sistema y se la diferenció de la "infraestructura"; se enumeraron los objetivos del trabajo y los temas de investigación. También se vieron los problemas asociados a una arquitectura incorrecta y los inconvenientes que surgen cuando la gestión de la arquitectura no es la adecuada.

En los capítulos siguientes se verá que el proyecto de Cientópolis tenía dos problemas: por un lado, una arquitectura mal diseñada, por lo que sufría de todas las deficiencias y problemas descritos anteriormente. En segundo lugar, la gestión de esa arquitectura tampoco era óptima, exhibiendo varios de los problemas mencionados anteriormente en este capítulo. La arquitectura propuesta en este trabajo resuelve el primer problema; mientras que la arquitectura programada resuelve el problema de la arquitectura mal gestionada.

El capítulo siguiente introduce el caso de estudio, Cientópolis, y la metodología usada para encarar la implementación de su nueva arquitectura.

Capítulo 2

Caso de estudio y metodología utilizada

La poesía tal vez se realza cantando cosas humildes.

Miguel de Cervantes Saavedra.
(1547-1616)
Escritor español

En la primera sección de este capítulo se explica qué es la ciencia ciudadana y se describe el proyecto Cientópolis, que es el caso de estudio de esta tesis. Luego, en la segunda parte del capítulo, se detalla la metodología seguida en este trabajo para encontrar la arquitectura más acorde a las necesidades de Cientópolis.

2.1. Caso de estudio: Cientópolis

2.1.1. Ciencia ciudadana

Muchos proyectos científicos requieren de un gran poder analítico que es tecnológica o financieramente inviable usando computadoras. Esos proyectos dependen de la gente *común* (ciudadanos) para analizar los datos. Por ejemplo, el proyecto SETI@home[1] depende de que los usuarios *donen* parte de sus ciclos de CPU ociosos para analizar porciones del espacio fotografiadas con telescopios. Los ciudadanos no se involucran directamente, sino que prestan recursos, sus computadoras en este caso.

Por otra parte, el proyecto The Plastic Tide (Zooniverse, s.f.), alojado en Zooniverse[2], requiere de interacción humana. Como explica en su página: "Etiquetando plásticos y basura en las imágenes tomadas con nuestro *drone*, estás enseñando a nuestro programa a autodetectar, medir y monitorear plásticos para ayudar a los investigadores a responder cuánto del 99 %[3] del plástico termina en nuestras playas"(Zooniverse, s.f.).

Si bien el término "ciencia ciudadana" es de reciente acuñación, la idea de personas no científicas *haciendo* ciencia es bastante antigua. De hecho, no fue hasta el siglo XX que la ciencia comenzó a ser dominada por instituciones científicas, gobiernos y academias. Para mayor información se recomienda leer el artículo Finalizing a Definition of "Citizen Science" and "Citizen Scientists" (OpenScientist, 2011).

[1] http://setiathome.berkeley.edu/

[2] Zooniverse es una plataforma donde se alojan varios proyectos de ciencia ciudadana (https://www.zooniverse.org/)

[3] Según las estimaciones hechas por los responsables del proyecto The Plastic Tide, se vierten a los océanos 8 millones de toneladas de plástico al año. Sin embargo los investigadores sólo pueden dar cuenta del 1 % de esa cantidad, de ahí que se pregunten ¿Adonde va el otro 99 % del plástico?

2.1.2. Proyecto Cientópolis

Cientópolis es un proyecto de ciencia ciudadana iniciado en 2015 por un grupo de investigadores del LIFIA de la UNLP, que en los últimos años creció hasta convertirse en el marco de trabajo de una comunidad mucho más amplia que involucra a investigadores y voluntarios de distintos países de Latinoamerica. Su objetivo es desarrollar comunidades, capacidades y tecnología para el avance de la ciencia ciudadana y ciencia abierta.

Al momento de escribir estos párrafos el proyecto cuenta con varios subproyectos que se encuentran en diversos grados de completitud. Es importante comprender que existe una relación de sinergia entre estos subproyectos y la arquitectura: por un lado estos sistemas imponen determinadas condiciones sobre la arquitectura, ya sea en la forma de propiedades a cumplirse, como puede ser la *performance* esperada; o en la forma de servicios a consumir, como un subsistema de *logging* con reglas bien definidas. Por otra parte, y aprovechando que varios de estos proyectos aún están en la etapa de diseño, la arquitectura puede plantear determinadas pautas de diseño e implementación como, por ejemplo, la estandarización de un mecanismo para acumular estadísticas de uso.

A continuación se mencionan los subproyectos de Cientópolis:

Runaway stars

Cientos de imágenes de posibles estrellas fugitivas (Runaway stars) necesitan ser analizadas por voluntarios en busca de características que permitan confirmarlas o descartarlas. Este proyecto es liderado por astrónomos de la Facultad de Ciencias Astronómicas y Geofísicas de la UNLP[4]. Están trabajando en este proyecto Agustín Kanner, Giuliano Dela Penna y Cintia Peri.

[4] https://www.cientopolis.org/portfolio/runaway-stars/

Galaxy Conqueror

El proyecto Galaxy Conqueror trata de un juego que tiene como objetivo "asistir a los investigadores identificando potenciales galaxias en una imagen del cielo conocida como la Ventana de Puppis"[5].

El equipo de trabajo detrás de Galaxy Conqueror está integrado por Juan Ignacio Yañez, Matías Celasco y Roberto Gamen.

Spotters

Spotters es un generador de aplicaciones de observación en la vía pública. El objetivo de este tipo de aplicaciones es lograr que los voluntarios marquen distintos puntos de interés para una determinada temática en el mapa, sirviéndose de Google Maps y StreetView[6].

El equipo que trabaj en Spotters está integrado por Ezequiel Claramunt y Agustín Marisi.

Samplers

Samplers, a cargo de Laura Lus y Javier Ramirez, es un *framework* para el muestreo móvil orientado a proyectos de ciencia ciudadana. En principio abarca aplicaciones que tomen imagenes, videos y audio geoposicionados. A futuro podría incorporar otro tipo de sensores. Se trata de un *framework* para aplicaciones en dispositivos móviles, por lo que impone pocas restricciones a la arquitectura de Cientópolis[7].

2.2. Metodología

Esta sección detalla la metodología aplicada al caso de estudio. El primer paso consistió en recopilar información acerca de la arquitectura original de Cientópolis, descrita en el Capítulo 6. Se analizó la información y se obtuvo un listado de falencias y deudas técnicas exhibidas por la arquitectura original.

[5] https://galaxyconqueror.cientopolis.org/ y https://www.cientopolis.org/portfolio/galaxy-conqueror/

[6] https://www.cientopolis.org/portfolio/spotters/

[7] https://www.cientopolis.org/portfolio/samplers/

Por medio de encuestas a los desarrolladores (que forman parte del grupo de los *stakeholders*), se obtuvo un listado de requerimientos funcionales y no funcionales, como qué lenguajes de programación se deben soportar o las versiones que se deben usar de los distintos motores de bases de datos.

Además, mediante reuniones con los directivos de Cientópolis, se elicitaron más requerimientos funcionales y no funcionales, surgiendo nuevas exigencias para la arquitectura.

Luego, se propusieron tres arquitecturas diferentes y se utilizó el método LAAAM para evaluar la relación entre las distintas arquitecturas y la calidad esperada del sistema. Cabe aclarar que en los tres casos se trata de arquitecturas perfectamente viables; es el método el que indica cuál de las tres se ajusta mejor a las necesidades de los *stakeholders*.

Por último, se implementó la arquitectura que más se acercó a las necesidades de los *stakeholders*. Este paso involucró el *desarrollo del código* usado para instanciar la arquitectura y el *desarrollo de la estrategia de gestión* de dicha arquitectura.

2.3. Resumen

En este capítulo se presentó el concepto de Ciencia Ciudadana, el proyecto Cientópolis y los subproyectos de trabajo que forman parte de Cientópolis. Es a estos subproyectos que la arquitectura debe brindarle sus servicios.

En la segunda parte del capítulo se explicó la metodología seguida para encontrar la arquitectura que mejor se adapta a las necesidades de Cientópolis.

El siguiente capítulo detalla las propiedades que caracterizan una arquitectura de sistemas, explica la relación entre arquitectura y calidad y se extiende sobre los patrones y estilos de arquitecturas.

Capítulo 3

Arquitecturas de software

> Lo más difícil de aprender en la vida es qué puente hay que cruzar y qué puente hay que quemar
>
> *Bertrand Russell*

En este capítulo se profundiza sobre las propiedades por medio de las cuales se puede evaluar una arquitectura. Por ejemplo, la **seguridad** es una propiedad de la arquitectura y distintas arquitecturas propuestas pueden variar en el grado de seguridad que presentan.

También se verá que el grado de **calidad** exhibido por una arquitectura está relacionado con la medida en que la arquitectura satisface cada una de las propiedades descritas a continuación.

3.1. Propiedades de una arquitectura de sistemas

Se puede evaluar una arquitectura según el grado en que cumple con determinadas propiedades de la misma forma en que se hace con otros artefactos de *software*. Existen varios mecanismos para agrupar estas propiedades, como FURPS (Grady, R. & Casswell, D., 1987). El modelo FURPS sirve para clasificar atributos relacionados con la calidad del *software* y fue desarrollado por Hewlett-Packard a fines de los años '80s. En particular las siglas hacen referencia a los siguientes atributos: Functionality, Usability, Reliability, Performance y Supportability.

Otra enumeración de atributos relacionados con la calidad de un *software* está dada por la ISO 9126 (ISO/IEC 9126, s.f.).

El objetivo de esta sección es mencionar y ejemplificar estas propiedades, dado que serán utilizadas a lo largo de este trabajo para "medir" la nueva arquitectura. A continuación se enumeran las propiedades involucradas en el modelo FURPS y luego se hacen algunos comentarios sobre la ISO 9126.

Conjunto de características o *feature set*: se refiere al conjunto de funcionalidad y servicios brindado por la arquitectura. Por ejemplo, servicio de mensajería, centralización de logs, etc.

Usabilidad: se refiere a cuán usable es la arquitectura. Es, quizás, de los conceptos más ambiguos y más difíciles de precisar. Sin embargo, existen maneras de obtener una idea de cuán "usable" es una arquitectura. Algunos ejemplos:

- Consistencia en el nombre de las componentes, en los errores, en los mensajes de salida de los servicios.
- Mensajes de error bien definidos.
- Uso de herramientas de automatización.
- Implementación de tests de la infraestructura.

Performance: da una idea del desempeño de la arquitectura bajo diferentes ambientes o condiciones. Un dato importante es la cantidad máxima de usuarios concurrentes bajo ciertas condiciones. Otras métricas interesantes podrían ser el peor tiempo de respuesta, el consumo medio de recursos o la medida en que se degrada el sistema conforme se lo utiliza.

Fiabilidad o *Reliability*: indica la probabilidad de que la arquitectura funcione correctamente bajo ciertas condiciones y durante un determinado periodo de tiempo. Existen algunas métricas bastante utilizadas, como son la frecuencia, la severidad y el tiempo medio entre fallos, las cuales se utilizan para calcular la *recuperabilidad* y *predictibilidad* de toda la arquitectura.

Instalabilidad: da una idea de cuán fácil o difícil es instanciar la arquitectura en un ambiente específico. La principal manera de incrementar este índice es mediante el uso de tests que aseguren que todas las componentes quedaron correctamente instaladas y configuradas.

Mantenibilidad: indica el esfuerzo empleado en realizar tareas de mantenimiento, el cual puede ser preventivo, perfectivo, adaptativo o correctivo. Una documentación actualizada y completa facilita las tareas de mantenimiento, así como el *testing* constante desde el inicio del proyecto.

Documentación: se trata principalmente de indicar la calidad de la documentación. Algunas cuestiones a considerar: la consistencia de la documentación, qué tan actualizada se encuentra, si está disponible en varios idiomas, si todos los servicios ofrecidos se encuentran debidamente documentados, etc.

Adaptabilidad o portabilidad: da una medida del esfuerzo empleado para adaptar la arquitectura a un nuevo ambiente. Requiere también de una batería de tests numerosa para verificar que todas las componentes operan correctamente. El uso de estándares, protocolos y lenguajes bien definidos ayuda en el proceso de adaptación a otros contextos de ejecución.

Auditabilidad: esta propiedad plantea la necesidad de auditar los sistemas. Consiste en asegurar mecanismos para el rastreo de decisiones de diseño y para rastrear las operaciones realizadas en las componentes de la arquitectura, como puede ser el *login* de un usuario o los comandos que haya ejecutado. En algunos casos el contexto legal torna imperativas determinadas medidas, aunque no es el caso de Cientópolis.

Disponibilidad: la proporción de tiempo que una arquitectura se encuentra brindado servicio. Existen varios conceptos asociados, como son:

- Alta disponibilidad (HA, por sus siglas en inglés), asociado a su vez con la redundancia.
- ETR o Tiempo estimado de recuperación.
- Porcentaje de disponibilidad. Por ejemplo, un porcentaje de 99 % anual indica que el sistema puede estar 3,65 días sin brindar servicio[1].
- También implica la existencia de una política de *backups* y un plan de contingencia.

Configurabilidad: indica el esfuerzo para configurar el sistema una vez instalado. Una solución automatizada ataca de frente este item.

Extensibilidad: indica el esfuerzo requerido a la hora de crecer el sistema. Aquí "crecer" indica dar más funcionalidad o agregar elementos que antes no se encontraban en la arquitectura. Los tests son esenciales, para que al agregar funcionalidad se compruebe que todo lo demás sigue funcionando. Por ejemplo, agregar soporte para un nuevo motor de bases de datos es una manera de extender la arquitectura.

Escalabilidad: habla sobre el esfuerzo requerido para "escalar" la arquitectura. La diferencia con la extensibilidad radica en que en este caso no se agregan conceptos nuevos a la arquitectura, sino que se agregan más recursos. Por ejemplo, se agrega más memoria RAM y la arquitectura escala *verticalmente*; o se agregan más nodos web (servidores encargados de procesar los requerimientos HTTP), y la arquitectura escala *horizontalmente*. Por diseño, se buscará que la arquitectura escale horizontalmente, dado que es la manera de proveer un escalado virtualmente infinito; mientras que un escalado vertical se hace principalmente agregando recursos de hardware.

Administrabilidad: indica el esfuerzo involucrado a la hora de administrar la arquitectura. En este caso se vuelve imperativo disponer de herramientas de monitoreo, análisis de logs, estadísticas y alertas.

[1] https://en.wikipedia.org/wiki/High_availability#Percentage_calculation

Modularidad: por definición, es el grado en que las componentes de un sistema pueden ser separadas y recombinadas. Esto implica módulos bien definidos y estándares para que se comuniquen entre sí. La modularización es lo que permitirá a la arquitectura escalar, crecer y ejecutarse en ambientes heterogéneos. Se trata de un concepto muy ligado a la reusabilidad.

Reusabilidad: indica el grado en que se reusan las componentes de un proyecto. No se refiere sólo a artefactos de software, como son las librerías, sino también a elementos de documentación, tests, scripts de automatización, etc. Como se indicó previamente, se trata de un concepto íntimamente ligado a la modularidad.

Seguridad: indica el grado de protección con que cuenta la arquitectura, así como también la cantidad de medios de protección brindados por ésta a sus aplicaciones. Algunas herramientas o mecanismos que incrementan el grado de seguridad de un sistema son:

- Uso de TLS.
- Accesos por SSH con un mecanismo de clave pública/privada.
- Uso de IDS, por ejemplo Snort[2].
- Herramientas de monitoreo, análisis de logs y alertas.

Estabilidad: es una medida de la cantidad de cambios sufridos por la arquitectura. Indica la madurez de un proyecto: al comienzo se espera una gran cantidad de cambios en los requerimientos y la implementación, pero a medida que se entiende mejor lo que se debe implementar y el proyecto avanza, la cantidad de cambios disminuye. No se debe confundir con el concepto de fiabilidad revisado al inicio de esta lista.

Testeabilidad: indica el grado en que se puede *testear* o probar una arquitectura. Por ejemplo, una métrica interesante es el porcentaje de la arquitectura que se puede probar mediante tests automatizados.

[2] https://www.snort.org/

Como se dijo al inicio de esta sección, otra enumeración de atributos relacionados con la calidad de un software está dada por la ISO 9126. En ella, la calidad del software está clasificada según las siguientes características:

Fiabilidad: son los atributos que tratan sobre la capacidad que tiene el sistema de mantener su nivel de desempeño bajo determinadas condiciones durante un cierto período de tiempo.

Usabilidad: un conjunto de atributos que hablan sobre el esfuerzo necesario para utilizar el sistema.

Funcionalidad: los atributos que tratan sobre la existencia de determinadas funciones en el sistema y sus propiedades. Estas funciones son las mencionadas en la etapa de requerimientos.

Mantenibilidad: habla sobre el esfuerzo necesario para realizar algún cambio.

Eficiencia: el conjunto de atributos que establece la relación entre el nivel de desempeño y los recursos consumidos bajo ciertas circunstancias.

Portabilidad: entendida como el conjunto de atributos que permiten transferir el software de un entorno a otro.

Cada una de estas características tiene, a su vez, un conjunto de subcaracterísticas. Por ejemplo, una subcaracterística de fiabilidad es la tolerancia a fallos.

La ISO 9126 fue reemplazada por la ISO 25010 (ISO/IEC 25010:2011). En el capítulo introductorio de la nueva ISO se mencionan las diferencias con respecto a la ISO 9126. Aquí se enumeran los cambios relevantes en el contexto de este trabajo:

- La **Seguridad** se considera una característica en vez de una subcaracterística de la Funcionalidad.
- Introduce la **Compatibilidad** como una nueva característica.

No es el objetivo de esta sección describir exhaustivamente los modelos relacionados con la calidad del software, sino más bien brindar al lector un panorama acerca de todas las propiedades que se pueden considerar sobre un sistema, ya sea una aplicación o la arquitectura sobre la que yace.

En cuanto al presente trabajo, se toma como referencia las propiedades enumeradas por FURPS. Se basa esta decisión en la simplicidad que exhibe con respecto al modelo dado por la ISO 9126/25010:2011.

Restan, para finalizar esta sección, dos aclaraciones importantes. En primer lugar, cuando se habla de propiedades como fiabilidad o usabilidad, ¿a qué se aplican? Existen cuatro *artefactos* a los cuales se pueden aplicar esos términos (y la mayoría de los demás):

Arquitectura: aquí se entiende la arquitectura como cosa abstracta modelada.

Implementación de la arquitectura: en este caso se habla de la arquitectura instanciada (servidores de bases de datos en ejecución y proxies reversos configurados, por ejemplo) y el código necesario del lado del administrador para instanciar esa arquitectura (scripts, roles de Ansible[3], configuración de AWS[4], etc).

Procesos involucrados: se refiere a los procedimientos por medio de los cuales se crean instancias de la arquitectura, se da de alta el espacio para una aplicación, se describe la manera en que se debe documentar el código de una aplicación, etc.

Aplicaciones de Cientópolis: son los proyectos o aplicaciones desarrolladas para correr sobre la arquitectura aquí diseñada.

En el contexto de este trabajo, cuando se habla de propiedades y de calidad, el lector debe entender que se hace referencia a los dos primeros ítems, es decir, el diseño de la arquitectura y su implementación. La calidad de esos dos ítems es lo que se considera. Evaluar la calidad de los últimos dos ítems escapa al alcance de este trabajo.

Para concluir, se debe aclarar que ciertas propiedades se relacionan con la arquitectura en funcionamiento (como por ejemplo la administrabilidad); mientras que otras propiedades están ligadas al ciclo de vida de la arquitectura, como puede ser el caso de la modularidad. Se espera que el lector pueda reconocer por el contexto si en un determinado momento se evalúa la arquitectura en su ejecución o en su ciclo de vida.

[3] `https://www.ansible.com/`, se describirá su uso más adelante.
[4] `https://aws.amazon.com/`, se mencionará su uso más adelante

3.1.1. Relación entre la arquitectura y la calidad de un sistema

La arquitectura de un sistema es el mecanismo que se tiene para asegurar que ese sistema cumple con ciertos atributos relacionados con la calidad del sistema, como performance, fiabilidad, seguridad, etc. Es necesario notar que la arquitectura es un elemento clave y necesario, pero no suficiente. La arquitectura debe brindar todas las herramientas al software que sostiene, pero es también papel de ese software asegurar esos atributos. La arquitectura debe guiar al desarrollador dando documentación y herramientas. Estos requerimientos de calidad son el principal motor en el diseño de una arquitectura.

Existen, de todas maneras, algunos atributos que están más relacionados con la arquitectura que con el software que sobre ella se ejecuta. Por ejemplo, en la documentación de ATAM (*Architecture Tradeoff Analysis Method*) (Kazman, R., Klein, R. & Clements, P., 2000), se menciona que en sistemas grandes lograr las metas de calidad, como desempeño, disponibilidad y modificabilidad, depende más de la arquitectura que de la aplicación en sí o de las tecnologías asociadas a la aplicación, como lenguaje de programación o algoritmos).

3.2. *Patterns* de diseño de arquitecturas

Son raras las ocasiones en las que al iniciar un proyecto no se dispone de conocimiento previo sobre el cual apoyarse. Por ejemplo, no es habitual que se comience a desarrollar una aplicación web de cero, sino que se disponen de varios *frameworks* para agilizar el desarrollo. Más aún, los desarrolladores -e incluso los *frameworks* mismos- disponen de una amplia variedad de patrones de diseño para utilizar.

Cuando se plantea la arquitectura de un sistema ocurre algo similar: ya existen respuestas probadas en reiteradas ocasiones a los desafíos que debe encarar el arquitecto del sistema. En esta sección se realiza un repaso de los patrones en general y se mencionan los patrones de arquitectura más comunes, dando ejemplos concretos sobre su utilización. La razón de esta sección es que en el diseño de la arquitectura de Cientópolis se utilizan varios de estos patrones.

Christopher Alexander escribió en 1977, acerca de la Arquitectura, que "cada patrón describe un problema que ocurre reiteradas veces en nuestro entorno y luego describe el meollo de la solución a ese problema, de tal manera que se puede utilizar esta solución un millón de veces sin hacer lo mismo dos veces"(Alexander, C., 1977). En el ámbito de la Informática esa idea fue tomada por primera vez en 1995 por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides en su libro Desing Patterns (Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1995).

Existen varias ventajas en usar patrones:

Manejo del conocimiento: permite reusar conocimiento y preservarlo para proyectos futuros.

Control de la complejidad: se puede controlar la complejidad de la arquitectura siguiendo patrones bien definidos y probados.

Conocimiento en común: el diseño, desarrollo y documentación se realiza utilizando términos bien definidos y familiares. Además, se reducen el tiempo y esfuerzo involucrados en el proceso de transferencia de conocimiento y aprender la arquitectura.

Mitigación del riesgo: al utilizar patrones ya probados el riesgo de acabar con una arquitectura deficiente es menor que si se diseñara la arquitectura de cero.

Reducción de tiempo: el uso de patrones permite empezar a trabajar sobre material existente, acelerando los tiempos de implementación, lo cual podría traducirse en una reducción de costos.

Unos párrafos más arriba se explicó la relación entre arquitectura y calidad de un sistema. El uso de patrones de arquitectura permite entender de antemano las consecuencias sobre los atributos de calidad que exhibirá la arquitectura planteada.

En el contexto de este trabajo, un patrón de arquitectura se verá como la descripción de un conjunto de componentes y sus responsabilidades. Por ejemplo, el patrón de cliente-servidor establece dos componentes, la componente **cliente** y la componente **servidor**, y responsabilidades para cada uno de ellos (el cliente solicita un recurso y el servidor realiza alguna acción para satisfacer la petición del cliente).

Esta descripción se realiza de manera independiente a la tecnología usada. Por ejemplo, en vez de hablar en términos de Apache httpd[5], se habla del servidor web o servidor HTTP. Como sucede con los patrones de diseño, los de arquitectura también tienen un nombre, una descripción del problema, una descripción de la solución y las consecuencias:

Nombre: Al darle un nombre al patrón se tiene una manera rápida y sencilla de referirse a un conjunto de problemas, soluciones y consecuencias.

Problema: Explica el problema y su contexto. Da una idea de cuándo aplicar el patrón.

Solución: Describe los elementos que hacen al diseño, las relaciones entre ellos y sus responsabilidades.

Consecuencias: Se refiere a los resultados y *trade-offs* de aplicar el patrón.

3.2.1. Estilos de arquitectura

En Distributed Systems (Tanenbaum, A. & van Steen, M., 2007), Tanenbaum habla de "estilos de arquitectura" en vez de "patrones de arquitectura". Más aún, existe muchísimo material en donde se habla de patrón o estilo como si fueran sinónimos. En esta sección se intenta hacer una distinción entre ambos. Sin embargo, como se verá, en la mayoría de los contextos se pueden usar como sinónimos. Es por eso que a lo largo de este trabajo se utilizará el término "patrón" para hacer referencia tanto a patrones de arquitectura como a estilos de arquitectura.

George Fairbanks (Fairbanks, G., 2014) marca la diferencia entre "patrón" y "estilo" utilizando el paradigma de objetos: al hablar de "estilo" se hace referencia a los elementos que conforman el paradigma, como objetos, clases y mensajes; mientras que al hablar de patrones, se hace referencia a un uso determinado de esos elementos, cada uno llevando a cabo un determinado rol en determinado contexto. Ambas son abstracciones, pero el "estilo" *define el vocabulario* y los "patrones" aplican *restricciones*.

[5] Apache httpd es uno de los servidores web más utilizados en el mundo, https://httpd.apache.org/

Como se dijo más arriba y dado que la diferencia es bastante sutil, a lo largo de este trabajo se utilizará el término "patrón" para hablar tanto de patrones de arquitectura como estilos de arquitectura.

3.2.2. Ejemplos de patrones de arquitectura

En esta sección se comentan algunos de los patrones de arquitectura más habituales, dado que la mayoría son utilizados por las tecnologías involucradas en la arquitectura de Cientópolis. No es el objetivo detallarlos exhaustivamente, sino mencionarlos y explicarlos con brevedad. El lector puede encontrar un análisis mucho más detallado en Patterns of Enterprise Application Architecture (Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R. & Stafford, R., 2002) y en Distributed Systems (Tanenbaum, A. & van Steen, M., 2007).

Patrón de cliente-servidor

Citando nuevamente a A. Tanenbaum (Tanenbaum, A. & van Steen, M., 2007), "los procesos de un sistema distribuido se dividen en dos grupos. Un servidor es un proceso implementando un servicio específico [...]. Un cliente es un proceso que requiere un servicio del servidor enviando un requerimiento y esperando la respuesta del servidor".

Los protocolos HTTP y SSH -todos ellos ampliamente usados en Cientópolis- son ejemplo concretos de este tipo de patrón cliente-servidor. La Figura-3.1 muestra un ejemplo de conexión entre un navegador web (cliente) y un servidor web.

Patrón de capas

En el caso del patrón de capas, las componentes de un sistema se disponen apiladas unas encima de las otras. Si C_i es la capa i-ésima y C_{i+1} es la capa apilada encima, entonces C_{i+1} puede enviarle requerimientos a C_i y esperar una respuesta, pero no al revés.

Los ejemplos más conocidos de uso de este tipo de patrón son los modelos de red OSI y TCP/IP. En el ámbito de este trabajo se utilizará TLS sobre TCP/IP.

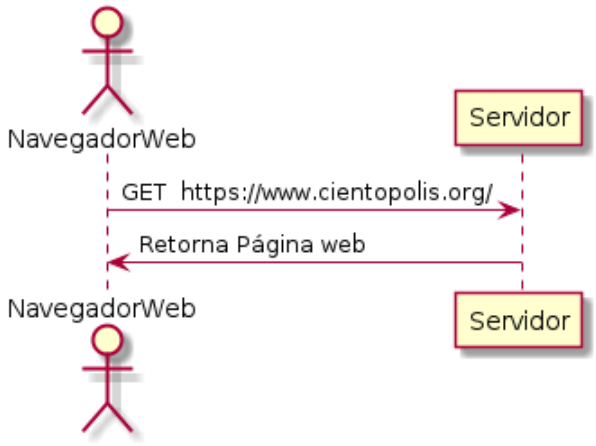

Figura 3.1: ejemplo de utilización del patrón de cliente-servidor

En el caso del modelo TCP/IP, TLS está por encima de la capa de transporte, posicionándolo como un protocolo de capa de aplicación; sin embargo, y dado que provee un medio de transporte cifrado, se lo suele ubicar entre la capa de aplicación y de transporte. En el caso del modelo OSI, se considera que TLS se posiciona en la capa de presentación. En la Figura-3.2 se puede ver la implementación de HTTPS siguiendo el patrón de capas.

Patrón de P2P

Las aplicaciones P2P (*Peer-to-peer*) se caracterizan por ser sistemas distribuidos donde todas las componentes del sistema tienen las mismas responsabilidades. Típicamente las redes P2P son bastante heterogéneas, en el sentido de que cada nodo tienen su propia configuración de recursos de procesamiento.

En el caso de Cientópolis, al momento de escribir estas páginas, no se utilizan tecnologías P2P. Sin embargo, se menciona esté patrón por completitud.

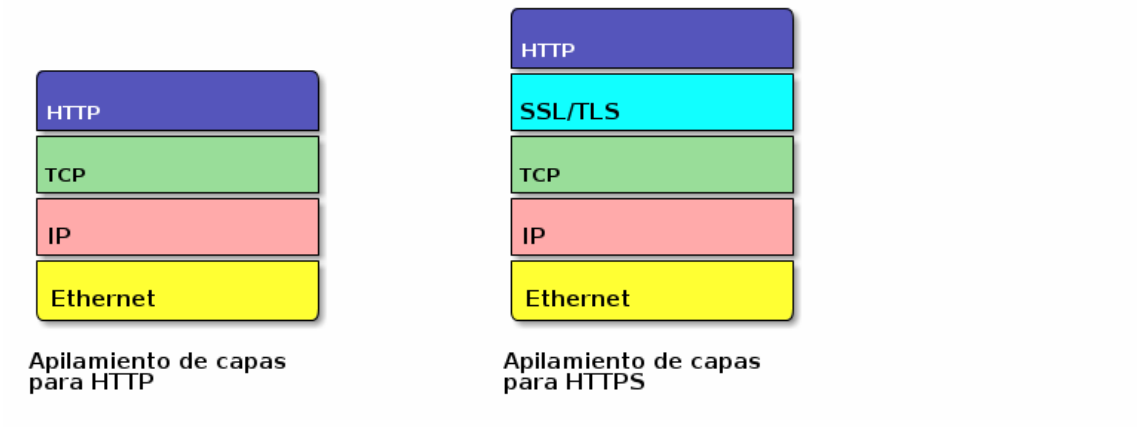

Figura 3.2: ejemplo de utilización del patrón de capas

Patrón *serverless*

Aún no existe un consenso sobre su definición. Se trata de código ejecutado en contenedores efímeros y sin estado en respuesta a algún evento. AWS Lambda[6] es la plataforma más conocida que implementa este tipo de servicio. Aquí se empieza a hablar de **FaaS**, *Functions as a service*. Para más información se recomienda el artículo de Mike Roberts, **Serverless Architectures** (Roberts, M. 2016). Al momento de escribir estas páginas no se utiliza este patrón en Cientópolis, pero se lo menciona por completitud.

3.3. Resumen

En este capítulo se describieron patrones de diseño estudiados y probados a lo largo de varios años y décadas, entre los que se encuentran los patrones de capas y cliente-servidor.

También se vio, al principio del capítulo, el modelo FURPS para hablar de las propiedades de una arquitectura de software y se mostró la relación existente entre dichas propiedades y la calidad resultante de la arquitectura.

[6] https://aws.amazon.com/lambda/

En el capítulo siguiente se mostrarán los métodos de evaluación de arquitecturas de modo que se pueda asegurar un determinado nivel de calidad.

Capítulo 4

Evaluación de arquitecturas

> Más vale enemigo cuerdo, que amigo loco
>
> *Refrán*

La arquitectura tiene un rol primordial en el cumplimiento de los requerimientos no funcionales y las metas de calidad. Sin embargo, es habitual que antes de empezar a trabajar, estos requerimientos no sean considerados exhaustivamente y si lo son, la mayoría de las veces el razonamiento detrás de las decisiones tomadas no queda descrito. Luego, es necesario un método para plasmar estas decisiones en la documentación o descripción de la arquitectura. Éste es el tema del presente capítulo, donde se habla de la evaluación de diferentes arquitecturas para llegar a aquella que es óptima en términos de la satisfacción de las necesidades de los *stakeholders*. De esta manera se vinculan los requerimientos no funcionales y las metas de calidad con las decisiones acerca de la arquitectura.

4.1. Métodos de evaluación de arquitecturas de sistema

En la actualidad existen diversos métodos de evaluación de arquitecturas de sistemas, siendo ATAM (*Architecture Tradeoff Analysis Method*) uno de los más antiguos y conocidos. Estos métodos se basan en priorizar determinadas cualidades deseables de la arquitectura, de modo que se asegura que dicha arquitectura responde a los requerimientos planteados por los *stakeholders*.

Un efecto secundario positivo es que agrupa a los *stakeholders* y requiere que se precisen bien los escenarios planteados. Por ejemplo, no sirve plantear escenarios tales como "el sistema debe ser tolerante a fallas", sino que se debe indicar algo como "El sistema debe tener un *uptime*[1] de 99,998 % anual". Y son los *stakeholders* quienes idean y priorizan estos escenarios.

Al evaluar la arquitectura se obtiene un mejor entendimiento de los *trade-offs* involucrados en el diseño de un sistema, que determinan la calidad de la arquitectura y del sistema. El objetivo de las metodologías de evaluación de arquitecturas es entender las consecuencias para la calidad del sistema que traen las decisiones arquitectónicas. Se debe entender que el empleo de estos métodos no asegura que se cumplan los requerimientos funcionales del sistema.

En el año 2002 M. Ionita describió los principales métodos de evaluación (Ionita, M. & Hammer, D., 2002), a los cuales debe agregarse uno más moderno, LAAAM. Así, los métodos más utilizados son:

- SAAM, Software Architecture Analysis Method
- ATAM, Architecture Trade-off Analysis Method
- CBAM, Cost Benefit Analysis Method
- ALMA, Architecture Level Modifiability Analysis
- FAAM, Family – Architecture Analysis Method
- LAAAM, Lightweight Architecture Alternative Assessment Method

[1] El *uptime* hace referencia al tiempo que el dispositivo ha estado operativo.

El método LAAAM es una variante "liviana" del método ATAM. ATAM explora las interdependencias de los atributos de calidad en una arquitectura y analiza qué tan bien una arquitectura satisface los requerimientos de calidad.

Estas metodologías buscan evaluar el impacto de decisiones arquitectónicas en los requerimientos de calidad. En todos los métodos es necesario definir escenarios. Los *stakeholders* diseñan los escenarios y sus propiedades. Es importante que los escenarios sean útiles y que la priorización de estos escenarios tenga sentido.

Según A basis for analyzing software architecture analysis methods (Kazman, R., Bass, L., Klein, M., Lattanze, T. & Northrop, L., 2005), existen cuatro criterios fundamentales que son de suma importancia para analizar y evaluar arquitecturas de sistemas:

Identificación de metas y contextos: es decir, ¿cómo se identifican y documentan los objetivos de la arquitectura y el contexto del análisis? El objetivo de todo análisis de arquitectura es evaluar una arquitectura y cómo se comporta frente a algún criterio, tales como futuros cambios, mayor carga de usuarios, etc. El contexto está dado por el estado actual de la arquitectura y las restricciones sobre el sistema.

Foco y propiedades de la evaluación: aunque el análisis de la arquitectura se hace de forma abstracta sin entrar en demasiados detalles de implementación, examinar todas las propiedades del sistema puede ser extenuante, por lo que cualquier análisis debe estar enfocado en uno o unos pocos aspectos a explorar. Es decir, ¿dónde se pone el foco y qué parte de la arquitectura se va a analizar?

Soportes o artefactos del análisis: un aspecto importante de todo método es que debe ser repetible, por lo que se deben tener herramientas que asistan y guíen a quien realiza el análisis. A su vez, el análisis produce resultados, que para ser comparables, también deben persistirse utilizando herramientas o artefactos determinados. Sin estas técnicas, documentos, ejemplos, plantillas, la *repetibilidad* del método se vería degradada.

Analizar los resultados: el análisis final cierra el círculo. Es decir, el método une las metas del primer paso, identificación de metas y contextos, con los resultados del punto anterior, soportes o artefactos del análisis.

El uso de alguna de estas metodologías reduce el riesgo de acabar con una arquitectura inadecuada.

4.1.1. Sobre los escenarios

Los escenarios sirven para guiar el diseño de la arquitectura y como casos de uso para medir su éxito o fracaso. Estos escenarios se basan en atributos que se relacionan con la calidad final del sistema, y ese nivel de calidad que debe alcanzar un sistema está en la mente de los *stakeholders*, por lo que es necesario que se involucren activamente en la creación de los escenarios.

Un escenario es una descripción corta de la interacción entre un *stakeholder* y el sistema. Existe una gran semejanza con los casos de uso utilizados en el desarrollo de software. La finalidad de un escenario es concretizar propiedades que al momento de iniciar el desarrollo son bastante vagas. También son muy útiles para entender y precisar cualidades que se manifiestan en tiempo de ejecución.

Como se menciona en *ATAM: Method for Architecture Evaluation* (Kazman et al., 2000), ATAM (Y por lo tanto LAAAM, por estar basado en ATAM) utiliza tres tipos de escenarios. En primer lugar se tienen los escenarios de caso de uso que "involucran el uso típico del sistema". Luego, los escenarios de crecimiento, que cubren cambios previstos en el sistema, por ejemplo, la expansión de usuarios. El tercer tipo está dado por los escenarios exploratorios, que trata con cambios estresantes para el sistema, por ejemplo, cambiar el sistema operativo de los servidores, o un corte de energía.

Si hay una meta de negocio que ningún escenario cubre, entonces los escenarios están incompletos; si hay un escenario que no se puede mapear a ninguna meta de negocio, entonces el documento con las metas de negocio está incompleto.

Para asegurarse que los escenarios se elicitan y documentan correctamente, todo escenario debe tener estas seis partes:

Estímulo: una condición que afecta al sistema.

Respuesta: la actividad del sistema que resulta como respuesta al estímulo.

Fuente del estímulo: la entidad que genera el estímulo.

Entorno: la condición o contexto en el cual ocurre el estímulo.

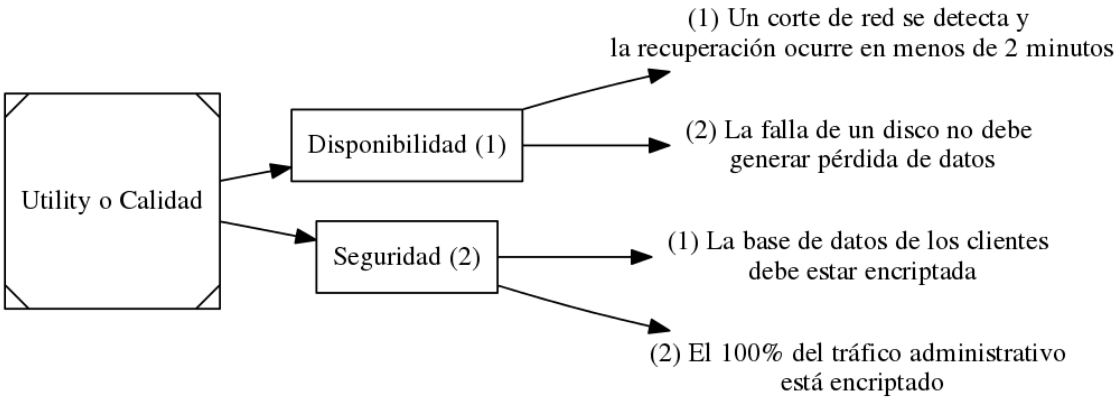

Figura 4.1: *utility tree* de ejemplo con escenarios de caso de uso. Se asume un ambiente de trabajo normal o típico.

Artefacto bajo estímulo: el artefacto que recibe el estímulo.

Medida de la respuesta: la medida según la cual se evaluará la respuesta del sistema.

Los escenarios se utilizan en conjunto con otra herramienta: los ***Utility Trees***, que traducen las necesidades del negocio en escenarios concretos que tratan con los atributos de calidad. Del *Utility tree* surge una lista priorizada de escenarios a satisfacer. En este contexto el término "*utility*" se puede pensar como un sinónimo de "calidad".

La Figura 4.1 muestra un *Utility Tree* a modo de ejemplo. Se puede apreciar que las dos necesidades de negocio principales son la disponibilidad y la seguridad. Sin embargo, el sólo mencionarlas no es suficiente, pues se trata de conceptos muy vagos. Es ahí que entran en juego los escenarios, con la finalidad de precisar cada una de esas dos necesidades.

En el ejemplo se muestran cuatro escenarios, dos para cada necesidad de negocio. Un escenario queda caracterizado por los siguientes ítems:

Contexto: se refiere al estado del ambiente cuando se aplica el escenario.

Estímulo: se trata de algo que le ocurre al sistema.

Respuesta: es la manera en que el sistema debe reaccionar.

Tomando como ejemplo el escenario "Un corte de red se detecta y la recuperación ocurre en menos de 2 minutos", se pueden ver las tres partes involucradas:

Contexto: está implícito el uso normal del sistema, dado que se trata de un escenario de caso de uso.

Estímulo: se produce un corte en la red.

Respuesta: el sistema se debe recuperar en menos de 2 minutos.

La participación de los *stakeholders* es de gran importancia al momento de pensar los escenarios. Como se dijo anteriormente, está en su mente el nivel de calidad pretendido para el sistema, pero también porque son ellos quienes asignan las prioridades. En el ejemplo de la Figura 4.1, los *stakeholders* consideran que la Disponibilidad es más importante que la Seguridad. A su vez, cada escenario tiene asignada una prioridad numérica.

La priorización de los atributos tiene dos fundamentos: por un lado, los recursos de desarrollo son limitados, por lo que se genera un *ranking* según la importancia de cada atributo. La segunda cuestión viene dada por algo ya mencionado: es común que determinadas propiedades deseadas vayan en contra de otras también deseadas. El clásico ejemplo es la performance y la seguridad: se puede agregar cifrado a un sistema para hacerlo más seguro, pero eso va en contra de la performance de ese sistema.

4.2. Elección del método de evaluación

Una evaluación rigurosa permite entender los requerimientos de calidad planteados por los *stakeholders*, la manera en que interactúan (*trade-offs*) y los riesgos asociados a las decisiones de diseño tomadas.

ATAM, en su rigurosidad, se convierte en un método "pesado" de llevar a cabo. Se requiere un nivel de participación de los *stakeholders* que no es posible en el proyecto Cientópolis. También produce una cantidad de artefactos inmanejable para el proyecto. LAAAM, por otra parte, es una alternativa *liviana* a ATAM, y es la elegida precisamente por esa cualidad. Por lo demás, LAAAM propone una manera similar de encarar el diseño de una arquitectura.

Entre las fortalezas de ATAM/LAAAM destacan:

- Mejor comprensión de la arquitectura por parte de los *stakeholders*.
- Mejora en la documentación de la arquitectura.
- Mejor documentación sobre las decisiones de diseño de una arquitectura.
- Mejora en la comunicación entre los *stakeholders*, debido a que ATAM/LAAAM sirve de lenguaje común entre todos los involucrados.
- Los *stakeholders*, mediante el uso de escenarios, deben precisar qué entienden ellos por calidad de la arquitectura.
- Se logra un consenso entre todos los *stakeholders* acerca del grado de calidad de la arquitectura.
- ATAM/LAAAM producen varios artefactos o productos que representan el proceso que hubo detrás de la toma de decisión. Estos artefactos sirven para justificar decisiones arquitectónicas.

Como se mencionó más arriba, se eligió LAAAM para este trabajo. Se justifica la decisión debido a varios aspectos importantes:

- LAAAM es un método ágil y liviano (*lightweight*) en lo que respecta a la carga de trabajo y la generación de documentos, en consonancia con la forma de trabajo del grupo Cientópolis.
- Como su predecesor, LAAAM es una metodología formal, que facilita a los *stakeholders* vislumbrar la arquitectura que buscan y entender y documentar las decisiones de diseño tomadas.
- La disponibilidad de los *stakeholders* para reunirse a debatir decisiones de arquitectura es limitada, por lo que no aplican métodos que consumen más tiempo, como ATAM mismo.
- LAAAM es liviano también en cuanto a la cantidad de entregables que genera, lo cual es bueno dado que no se dispone de personal para mantener demasiada documentación.

4.2.1. Método LAAAM

A continuación se describen los pasos para ejecutar LAAAM:

Paso 1 - Identificar los atributos de calidad: el primer paso consiste en identificar los atributos de calidad, aquellas propiedades que los *stakeholders* identifican como las necesidades básicas del sistema.

Paso 2 - Identificar los escenarios: en segunda instancia se deben acomodar los atributos de calidad elicitados en el paso anterior y armar el *utility tree* considerando los diversos escenarios.

Paso 3 - Priorización de los escenarios: los *stakeholders* deben priorizar y armar un *ranking* de escenarios.

Paso 4 - Asignar un peso a los escenarios: se puede utilizar cualquier método para calcular el peso de los escenarios. En este trabajo se usará una fórmula llamada Rank Order Centroids (ROC)[2], con la que se calculará un peso para cada escenario. La ecuación modela la manera en que las personas ven los *rankings* de prioridades, de modo que favorece los ítems "mejor ranqueados".

Paso 5 - Evaluación de las alternativas: por último se evalúa el comportamiento de los escenarios en cada alternativa, lo cual indica qué tan bien se adapta la alternativa de implementación a cada escenario. De aquí sale un número final indicando cuál de las alternativas tiene mayor puntuación. En otras palabras, la que mejor se lleva con cada escenario.

4.3. Resumen

En este capítulo se habló de las ventajas de utilizar algún método para evaluar arquitecturas de sistemas. Se enumeraron los métodos más conocidos y se profundizó sobre ATAM y LAAAM. El primero es un método precursor bastante *pesado* de ejecutar; el segundo, LAAAM, es una versión liviana del primero, más acorde a las metodologías ágiles usadas en el grupo de Cientópolis.

[2]https://www.vcalc.com/wiki/MichaelBartmess/Rank+Order+Centroid

También se habló acerca de los escenarios, su importancia y propiedades, debido a que son un elemento clave en LAAAM. Son los *stakeholders* quienes deben proponer y priorizar los escenarios.

Por último, se dieron las razones por las cuales se prefiere el método LAAAM y se mostró cuáles son los pasos genéricos para ejecutarlo.

ATAM y sus derivados son agnósticos en lo que respecta a la documentación. El capítulo siguiente trata sobre diferentes maneras de gestionar o documentar la arquitectura.

Capítulo 5

Gestión de arquitecturas

> Controlar la complejidad es la esencia de la programación.
>
> *Brian Kernigan*

En los capítulos previos se introdujo el concepto de "arquitectura" y se enumeraron varias propiedades por medio de las cuales pueden compararse varias arquitecturas. También se habló de métodos basados en escenarios para evaluar diferentes arquitecturas, de entre los cuales se eligió LAAAM.

Uno de los puntos fuertes de LAAAM, y en general de todo método basado en ATAM, es el hecho de que las decisiones de diseño quedan plasmadas en la documentación generada por el método. Sin embargo, si bien el método indica "qué" se debe hacer, no indica el "cómo". Este capítulo trata sobre dos maneras de documentar arquitecturas de sistema.

5.1. TOGAF - The Open Group Architecture Forum

TOGAF (Josey, A., 2011) y (Togaf 9.2) es un *framework* gratuito para el desarrollo de arquitecturas empresariales. La primera versión data de 1995 y está basado en TAFIM[1].

En la documentación de TOGAF se utiliza el término **empresa**[2] para denotar a la organización como un todo (tecnología, procesos, infraestructura) o a un dominio específico dentro de la organización. En ambos casos se considera que la arquitectura atraviesa diversos sistemas y grupos funcionales.

Una de las partes más importantes de TOGAF es ADM, Architecture Development Method, que describe un método para desarrollar y administrar todo el ciclo de vida de una arquitectura empresarial.

Una de las críticas más comunes a TOGAF es su poca practicidad. Svyatoslav Kotusev dice en su estudio Enterprise architecture is not TOGAF (Kotusev, 2016) que "[...] la mayoría de las recomendaciones de TOGAF son, por lo general, inaplicables [...]"[3]. Más aún, examinando la documentación se puede apreciar el gran volumen de pasos, productos y entregables requeridos para llevar a cabo esta metodología.

Se trata, pues, de un método poco *ágil* para el desarrollo y manejo de la arquitectura, va en contra de la necesidad del proyecto Cientópolis y es la razón principal por la cual se descarta como marco de trabajo para la elaboración de la arquitectura aquí presentada.

[1] Technical Architecture Framework for Information Management, desarrollado por el DoD (Departamento de Defensa de EE.UU.)

[2] Enterprise, en el original

[3] *[...] most TOGAF recommendations are usually found inapplicable [...]*

5.2. ISO 42010 - descripción de arquitecturas

Anteriormente se han mencionado varias propiedades mediante las cuales se puede evaluar una arquitectura. El grado con que una arquitectura satisface esas propiedades debe quedar documentado. Más aún, deben quedar documentadas las decisiones de diseño que llevaron a esa arquitectura. Esta documentación es la justificación por la cuál se tiene una arquitectura en particular y no otra. El problema es que muchas veces, como ocurre en el diseño de software, la arquitectura, y las decisiones involucradas en su diseño, no quedan bien documentadas, lo cual hace imposible *trazar* dichas decisiones.

Existe una gran cantidad de maneras de escribir o documentar las arquitecturas, y el primer estándar es relativamente nuevo, del año 2000[4]. La ISO 42010 sirve para describir y documentar arquitecturas de sistemas y brinda pautas generales que serán adaptadas a las necesidades del proyecto. Es decir, no se seguirá la ISO 42010 al pie de la letra, sino que se utilizará a modo de inspiración al momento de documentar la arquitectura. Existen dos razones principales para tomar esta decisión:

- La ISO 42010 es anterior al surgimiento de las metodologías "ágiles" de diseño de software y de las metodologías de *devops* para la administración de sistemas. En otras palabras, es demasiado "rígida" para el entorno de trabajo de Cientópolis.

- Cientópolis es un proyecto de pocas personas involucradas en el diseño de su arquitectura, por lo que gestionar todas la documentación sugerida por la ISO 42010 consumiría demasiado tiempo.

Además, dado que se utilizarán ambientes virtuales y técnicas de infraestructura como código, en cierta medida el mismo código servirá como documentación de la arquitectura de Cientópolis.

De todas maneras, donde sea provechoso, se generarán entregables que servirán para la comunicación entre los *stakeholders* y para capturar las decisiones de diseño cruciales.

[4]Notar que los sistemas informáticos existen como tales desde hace al menos 60 años

5.2.1. Vocabulario de la ISO 42010

La arquitectura habla de componentes y sus relaciones para reducir el problema a ítems más pequeños y manejables. La ISO 42010 es un estándar para documentar arquitecturas y servirá para explicar las relaciones entre dichas componentes.

El diseño del sistema se hace de acuerdo a varias vistas, de la misma manera en que para una casa se tienen varios planos, como eléctrico, de gas, de agua, etc.

El estándar no provee ninguna guía donde se recomienden los pasos a seguir para documentar una arquitectura. La ISO 42010 define varios conceptos. Aquí se enumeran los más relevantes en el contexto de este trabajo. Para una lista completa de definiciones se invita al lector a revisar la especificación de la ISO 42010. Se ha optado por incluir entre paréntesis el vocablo en original en inglés, tal cual aparece en la ISO.

Arquitectura (Architecture): propiedades o conceptos fundamentales de un sistema en un contexto dado expresados en sus elementos, relaciones y en los principios de su diseño y evolución[5].

Descripción de una arquitectura o AD (architecture description): Es un producto o entregable usado para expresar una arquitectura.

Vista (architecture view): es un producto o entregable que expresa la arquitectura desde la perspectiva de un *concern*[6] específico. Una vista expresa toda la arquitectura y no sólo una parte.

Punto de vista (architecture viewpoint): es un producto que establece las convenciones para construir, interpretar y usar una vista. Las convenciones pueden incluir el tipo de lenguaje a utilizar, los tipos de modelos, las reglas de diseño, las técnicas de análisis, etc.

Concern o asunto (concern): interés en el sistema relevante a uno o más stakeholders.

[5]Notar que esta definición es similar a la dada por Bijlsma, pero agrega la idea de **evolución** de la arquitectura.

[6]En este texto se utilizará el término en inglés.

Entre los *concerns* mencionados por el estándar se encuentran[7]: funcionalidad, propiedades del sistema, limitaciones conocidas, desempeño, consumo de recursos, seguridad, complejidad, costo, accesibilidad de los datos y conformidad con las regulaciones.

El estándar está pensado para trabajar con descripciones de arquitecturas y no con arquitecturas. Hace la siguiente distinción o aclaración: una descripción de arquitectura es un producto, mientras que una arquitectura es abstracta y consiste de conceptos y propiedades. El estándar tampoco especifica ningún formato a utilizar, así como tampoco prescribe ningún método o proceso para producir las descripciones.

5.3. Resumen

En este capítulo se presentaron dos marcos de trabajo para documentar arquitecturas de sistemas. Se habló brevemente de TOGAF y se explicaron las razones por las cuales no será utilizado. Luego se habló de la ISO 42010 y se explicó que, aunque no se seguirá con rigurosidad, sí se utilizará como guía al momento de componer los entregables frutos de ejecutar LAAAM.

El siguiente capítulo detalla la arquitectura actual de Cientópolis, sus falencias o deudas técnicas y las motivaciones para cambiarla.

[7]Para la lista completa puede revisar el estándar.

Capítulo 6

Evaluación de la arquitectura existente y estrategias de gestión de la misma

El software es un gas: se expande hasta llenar su contenedor

Nathan Myhrvold

6.1. La arquitectura existente

La definición de una arquitectura se realiza con el objetivo de alcanzar determinadas propiedades relacionadas con la calidad del producto final, como pueden ser *modificabilidad*, *performance* o *seguridad*, por poner sólo algunos ejemplos[1].

[1]Más adelante se verá que no alcanza con indicar que un sistema "tenga buena performance", si no que se debe hilar más fino y ser más preciso a la hora de definir los requerimientos de la arquitectura.

Más aún, todo sistema tiene una arquitectura, aunque no se haya planeado, sea desconocida y nunca se haya documentado. El actual diseño de la arquitectura de Cientópolis no es producto de un análisis formal, sino que fue tomando forma a medida que el proyecto de ciencia ciudadana avanzaba y surgían nuevos desafíos y necesidades. En este capítulo se presenta la arquitectura que tenía Cientópolis antes de iniciar este trabajo y se enumeran las falencias que se encontraron. Luego se mencionan las motivaciones existentes para cambiar esa arquitectura y por último se comenta la nueva funcionalidad esperada para la nueva arquitectura.

Para evaluar la arquitectura actual de Cientópolis y sus subproyectos se creó una encuesta para ser completada por los desarrolladores. En esta sección se comenta la arquitectura existente de Cientópolis utilizando los resultados de esta encuesta como principal medio de elicitación. La encuesta completa está en el Apéndice-A.

Todos los proyectos de Cientópolis residen en una máquina virtual Ubuntu 14.04 LTS, usando Proxmox 3.4.11 como plataforma de virtualización. La máquina virtual dispone de 1 core de 3092 Mhz, 1GB de memoria RAM y 30GB de espacio en disco. La máquina se encuentra en una VLAN propia separada del resto de la infraestructura. Los datos estáticos (imágenes, hojas de estilo, etc) no se almacenan en ninguna CDN.

Se utiliza el software *open source* Bacula para realizar las copias de resguardo o *backups* de las bases de datos MySQL. Es decir, el código y los archivos de datos (imágenes, documentos de texto, etc.) que no se encuentran en el motor de bases de datos (como archivos subidos por los usuarios) no se resguardan[2].

El ambiente de virtualización cuenta con dispositivos UPS (*Uninterruptible power supply*) y grupo electrógeno. Por otro lado, los discos del sistema operativo residen en el virtualizador y no en el storage, lo cual dificultaría cualquier proceso de migración en vivo. De todas maneras no hay implementado ningún esquema de tolerancia a fallos ni se cuenta con plan de contingencia escrito.

El aspecto físico, que incluye la duración de las UPS, existencia de grupo electrógeno y varios caminos para el ruteo, no afectan el análisis que se haga en este trabajo. Podría pensarse que ésos son servicios que la arquitectura consumirá o asumirá que existen; es decir, están por debajo de la arquitectura y, por ende, fuera del alcance de este trabajo.

[2]Aunque sí se hacen *snapshots* de la máquina virtual

Según el relevamiento, se utilizan apache2 y nginx como servidores web y como proxies reversos. En cuanto a las bases de datos, se utilizan PostgreSQL, MySQL, Redis y SQLite3.

El proyecto basado en Zooniverse en particular utiliza otras tecnologías adicionales, como Cellect y Kafka.

En cuanto a las aplicaciones, los lenguajes de programación utilizados del lado del servidor son Python, PHP, Ruby y NodeJS. Los frameworks usados son Django y Ruby on Rails.

El *despliegue*[3] de las aplicaciones se hace "a mano". Además, las componentes de Zooniverse quedan *desplegadas* utilizando Docker y Composer.

Antes de avanzar con las deudas técnicas de la arquitectura actual, es interesante hacer las siguientes menciones:

- PHP, aunque presente, es el menos utilizado. Aquí aparecen dos posibilidades: se puede migrar el poco código a otro lenguaje y eliminar PHP del proyecto o se lo puede dejar para que la arquitectura soporte PHP con el fin de ampliar el rango de programadores que la puedan usar. Se decidió incluir PHP para que los manejadores de contenidos de Cientópolis (MediaWiki y Wordpress) se puedan alojar en las mismas instancias.

- Hasta el momento no existen aplicaciones Java en Cientópolis. Surge la pregunta obligada de si se debe o no incluir Java en el diseño de la arquitectura. Aquí de vuelta aparece la finalidad de acaparar o no más programadores. Al momento de escribir este capítulo, la respuestas es negativa.

- Tampoco existen aplicaciones, hasta el momento, escritas en C, C++ o algún otro lenguaje compilado, por lo que no se considera darles soporte en la arquitectura diseñada.

[3] Traducción literal de "deploy". En algunos círculos también se acepta el término castellanizado "deployar".

6.2. Falencias o deudas de la arquitectura actual

En esta sección se detallan las falencias de la arquitectura actual de Cientópolis. Debe notarse que "falencia" es una propiedad que se desea y que la arquitectura no la provee actualmente. Por ejemplo, si la alta disponibilidad no es requerida, entonces la ausencia de esa propiedad en la arquitectura no debe considerarse como falencia.

A continuación se listan las falencias encontradas luego de relevar la arquitectura existente:

- Bajo grado de **modificabilidad**. No hay documentación y la arquitectura es monolítica. Esto implica que realizar cualquier cambio es costoso en términos de tiempo. Además, las modificaciones son más riesgosas, por la falta de documentación y por la ausencia de un ambiente de testing para la arquitectura.

- Alto costo para el desarrollador, dado que la plataforma no brinda ningún servicio, como puede ser autenticación centralizada, cifrado sobre TLS, etc. El desarrollador debe implementar todo lo que necesita e incluso configurar el servidor para que aloje su aplicación.

- La arquitectura actual no escala horizontalmente, afectando la performance. Se trata de una arquitectura con grado muy bajo de **escalabilidad**, lo cual pone un límite al crecimiento futuro que pueda experimentar el proyecto.

- Bajo grado de **administrabilidad**. No hay centralización de logs, documentación sobre la configuración de los servicios ni monitoreo.

- No se han llevado a cabo pruebas de estrés ni se han hecho análisis sobre el consumo de recursos, por lo que no es posible identificar los límites de **performance** ni indicar la cantidad máxima de usuarios concurrentes.

- No se lleva un log o bitácora de fallos, severidad de fallos, tiempo medio entre fallos. La arquitectura no fue pensada en términos de **recuperabilidad** y/o **predictibilidad**. No existe un plan de contingencia. Se puede decir que el criterio de administración actual es el de "mejor esfuerzo".

- No existen tests para verificar la correcta instalación de las componentes, lo cual dificulta la **mantenibilidad** e **instalabilidad** de la arquitectura. Más aún, como se menciona unas lineas más arriba, muchas veces son los desarrolladores quienes "ponen a punto" los servicios de los que depende su aplicación.

- La arquitectura actual no es modular y no se puede replicar con facilidad, lo cual afecta seriamente su **escalabilidad** e **instalabilidad**[4].

- La arquitectura actual no es reusable. Está implementada "a mano" y no cuenta con documentación que pueda ser reusable.

- La seguridad provista por la arquitectura es baja. Por ejemplo, no se garantiza que los servicios operan sobre TLS ni se restringe el acceso de administrador a sólo aquellos que tienen ese rol[5].

- Ausencia de ambientes de desarrollo, testing y producción. Como se dijo en párrafos anteriores, la ausencia de ambientes de desarrollo y testing para la arquitectura se traduce en un incremento en el costo de modificación de la arquitectura. El mantenimiento también se torna más engorroso y problemático. Tampoco dispone, el programador, de un ambiente de desarrollo bien definido, lo cual puede llevar a incoherencias entre los ambientes de desarrollo y producción.

6.3. Motivaciones para cambiar la arquitectura y nueva funcionalidad deseada

En esta sección se detallan los objetivos de la comunidad de Cientópolis que motivan el rediseño de la arquitectura y sirven de requerimientos para la nueva arquitectura.

[4]Esto último es muy importante si se considera que una de las directrices del proyecto es facilitar la arquitectura a otros organismos para que puedan alojar sus propios proyectos de ciencia ciudadana

[5]En otras palabras, todos los usuarios son administradores

La principal motivación es favorecer el crecimiento de Cientópolis, tanto en la cantidad de usuarios como de programadores de aplicaciones. Para satisfacer este fin, la administración de la arquitectura debe escalar fácilmente. Por ejemplo, debe llevar el mismo esfuerzo manejar una comunidad de cuatro o cinco desarrolladores que una de cincuenta.

Además, para atraer un espectro más amplio de desarrolladores, la arquitectura debe soportar varios *frameworks* y lenguajes de programación, así como también brindarle al desarrollador la mayor cantidad posible de herramientas.

También, continuando con los desarrolladores, se desea liberarlos de las tareas de administración y mantenimiento, como son: instalar y configurar los servicios, responder ante alertas del sistema, mantener actualizado el servidor, etc. Brindar servicios a los desarrolladores y administradores de Cientópolis, como estadísticas, buses de comunicación sincrónicos y asincrónicos y TLS por defecto es una manera de mejorar la productividad de este grupo de *stakeholders*. Como se mencionó anteriormente, en la arquitectura actual si el desarrollador desea utilizar alguna de estas herramientas, debe instalarlas, configurarlas y mantenerlas por su cuenta[6].

Por otra parte, los administradores de Cientópolis comenzaron a ver la arquitectura como un elemento más que se puede brindar a la comunidad, de modo que ésta la instancie e implemente sus propias aplicaciones de ciencia ciudadana. Esta es una motivación interesante que está íntimamente relacionada con la **instalabilidad** de la arquitectura y que, como se verá más adelante, es una de las razones principales por las cuales se decidió utilizar técnicas de infraestructura como código.

[6]Esta práctica deriva generalmente en errores cuando un desarrollador configura el servidor compartido de forma tal que se vuelve incompatible con otras aplicaciones previamente instaladas y funcionando.

Existe, también, una motivación financiera: Amazon dona recursos de su nube durante un año para que puedan usarse en el marco del proyecto Cientópolis. Esto propicia el deseo de migrar del actual servidor en Proxmox a la nube de Amazon. Migrar de la solución actual basada en Proxmox a la nube de Amazon, permitirá experimentar con los conceptos de infraestructura como código, a la vez que se mejorará la **instalabilidad** y **reusabilidad** de la arquitectura. Todo esto sin perder de vista que la arquitectura también deberá poder instanciarse en un ambiente de virtualización basado en Proxmox, dado que cuando se venza el crédito en Amazon, se deberá volver a Proxmox.

Otra cuestión importante a tratar es la descentralización de la arquitectura pero manteniendo la administración centralizada. La correcta separación de los elementos que componen la arquitectura, como servidores web, bases de datos, caches, etc. permitirá modificar, escalar y reusar componentes de la arquitectura de una manera mucho más rápida y predecible.

Por último, se desea mejorar la política de copias de resguardo o *backups* y comenzar a monitorear los servicios y aplicaciones instalados. De esta manera se espera mejorar los tiempos de respuesta ante fallas del sistema, disminuyendo a la vez la posibilidad de pérdida de datos. También se pretende disponer de un panorama más exacto a la hora de dimensionar la arquitectura (memoria RAM necesaria, tamaño de las bases de datos, etc).

6.4. Gestión actual de la arquitectura

En cuanto a la gestión de la arquitectura actual, también se han encontrado varias falencias que a continuación se describen.

Documentación inadecuada: no existen estándares, formales o no, que los integrantes del proyecto sigan para escribir la documentación. Como resultado, cada usuario configura sus aplicaciones según su propio criterio.

Roles indefinidos: no existe una clara separación de roles, lo que posibilita que los desarrolladores de las aplicaciones pueden ingresar al servidor y convertirse en administrador o *root*[7].

[7] El usuario *root* tiene todos los privilegios administrativos en el servidor.

Inexistencia de monitoreo y alertas: el monitoreo y las alertas existentes no son específicas para las aplicaciones. Por ejemplo, se monitorea que el servidor web y el motor de bases de datos estén funcionando, pero no que una aplicación tenga conexión con su base de datos.

Política de logs inexistente: no existe una política para tratar los logs del sistema y de las aplicaciones. Se utilizan los parámetros por defecto del sistema y de los *frameworks* usados en el desarrollo de cada aplicación.

Seguimiento de cambios: no existe un mecanismo que permita realizar el seguimiento de los cambios de la arquitectura. Es decir, una manera de ver todos los estadios por los que pasó la arquitectura, con la posibilidad de *volver atrás* un cambio.

Muchas de estas cuestiones se pueden resolver de forma relativamente fácil, mientras que otras pueden requerir una solución más compleja. Más allá del problema en particular, sería deseable maximizar el uso de herramientas de automatización en lo referente a la gestión de arquitecturas, de modo que apenas muy pocas cuestiones terminen dependiendo de una persona ejecutando una tarea de forma manual.

Por ejemplo, agregar el monitoreo de una aplicación se puede hacer de forma automática en el momento en que se crea el espacio en el servidor para esa aplicación; por otro lado, el uso de herramientas de automatización como Ansible[8] o Capistrano[9] permiten minimizar la documentación para los administradores.

6.5. Resumen

En este capítulo se dio una descripción de la arquitectura actual de Cientópolis y se describieron las deudas técnicas encontradas. También se explicaron las motivaciones para cambiarla y la nueva funcionalidad requerida.

[8] https://www.ansible.com/
[9] http://capistranorb.com/

Si bien no es difícil diseñar una arquitectura superadora, sí se presenta como un tarea más compleja implementarla y *desplegarla* en la nube de Amazon, a la vez que se pretende mejorar su **instalabilidad**, **mantenibilidad** y manejar la compatibilidad con Proxmox.

El capítulo siguiente muestra las tres opciones propuestas para la nueva arquitectura, utilizándose LAAAM para evaluarlas y elegir aquella que más se acerca a las necesidades de los *stakeholders*.

Capítulo 7

Propuesta, evaluación y selección de una nueva arquitectura

> No preguntemos si estamos plenamente de acuerdo, sino tan sólo si marchamos por el mismo camino.
>
> *Johann von Goethe*

En los capítulos previos se comentó la arquitectura actual de Cientópolis y se enumeraron las falencias exhibidas en su diseño, implementación y gestión; se habló de las arquitecturas en general y sus propiedades; se explicó también que existen métodos para evaluar arquitecturas y descubrir cuál de ellas se adapta mejor a los requerimientos de los *stakeholders* y de estos métodos se eligió LAAAM.

En este capítulo se presentan tres arquitecturas alternativas para Cientópolis. En los tres casos se trata de arquitecturas totalmente viables y la decisión de cuál implementar se hará según la puntuación que cada una obtenga al aplicarse el método LAAAM.

7.1. Evaluación de la arquitectura según el método LAAAM

En el capítulo-4 se mencionaron varios métodos para evaluar y comparar arquitecturas de sistemas y de entre todos se eligió LAAAM como método de evaluación para la nueva arquitectura de Cientópolis.

7.1.1. Principios para la arquitectura de Cientópolis

Una buena práctica recomendada por TOGAF es documentar los principios que van a actuar como guías a la hora de pensar la arquitectura. Dice TOGAF, acerca de los requerimientos de arquitectura que "[...] reflejan un nivel de consenso en la organización [...]. Los principios gobiernan los procesos propios de la arquitectura, afectando el desarrollo, mantenimiento y uso de la arquitectura".

Como se mencionó en el capítulo correspondiente, es bastante engorroso trabajar con TOGAF. Por ejemplo, según TOGAF, un principio debería consistir de nombre, declaración, fundamento e implicaciones. También habla sobre las cualidades que debe presentar un principio de arquitectura. Está más allá del alcance de este trabajo listar los principios de arquitectura de una manera tan estricta como lo hace TOGAF. Por el contrario, se elije una manera más informal de hablar acerca de estos principios:

Seguridad de los datos: los datos de la arquitectura y de las aplicaciones deben estar disponibles sólo a los usuarios autorizados.

Control de la deuda técnica: la deuda técnica debe ser controlada y contemplada en un plan de mejora continua.

Preferencia por las licencias *open source*: el software, protocolos y formatos usados deben estar publicados bajo licencias libres, prefiriéndose la licencia BSD.

Preferencia por los estándares: el software, protocolos y formatos utilizados deben respetar los estándares correspondientes.

7.1.2. Descripción de la arquitectura propuesta

Si bien se trata de aplicaciones web relativamente sencillas, el hecho de que se hayan escrito en varios lenguajes y usen diversos motores de bases de datos, junto con la escasez de recursos para administrar la arquitectura y que se desea una fácil instalación y mantenimiento de la arquitectura, hace más difícil un diseño que pueda satisfacer a todas las partes.

La Figura 7.1 muestra la arquitectura genérica de la solución. A continuación se plantean modificaciones a esta arquitectura genérica, dando por resultado tres alternativas, que son las alternativas que entran en competencia, usando LAAAM para deducir cuál de estas tres alternativas es la que mejor se adecúa a los requerimientos de los *stakeholders*.

En todos los casos se utilizarán técnicas de infraestructura como código para gestionar estas alternativas, independientemente de cuál sea la alternativa resultante de aplicar LAAAM.

Leyenda de los diagramas de arquitectura

En las secciones siguientes se muestran las alternativas de arquitectura propuestas para Cientópolis, pero antes de presentarlas, es conveniente explicar el significado de la leyenda que aparece en cada diagrama.

Servicio de AWS: indica servicios provistos por AWS. En el caso de la instanciación de la arquitectura en Proxmox, estos servicios se deben instalar y configurar. Pero en el caso de la instanciación sobre AWS, estos servicios ya están provistos.

Red o sistema externo: hace referencia a servicios o entidades externas, de las cuales no se tiene ningún tipo de control.

Cloud VM: se refiere a las máquinas virtuales en Proxmox o instancias en AWS.

Requiere backup: indica que esa entidad requiere de copia de resguardo.

Tráfico externo-interno: indica el tráfico de red entre la arquitectura de Cientópolis y las entidades externas.

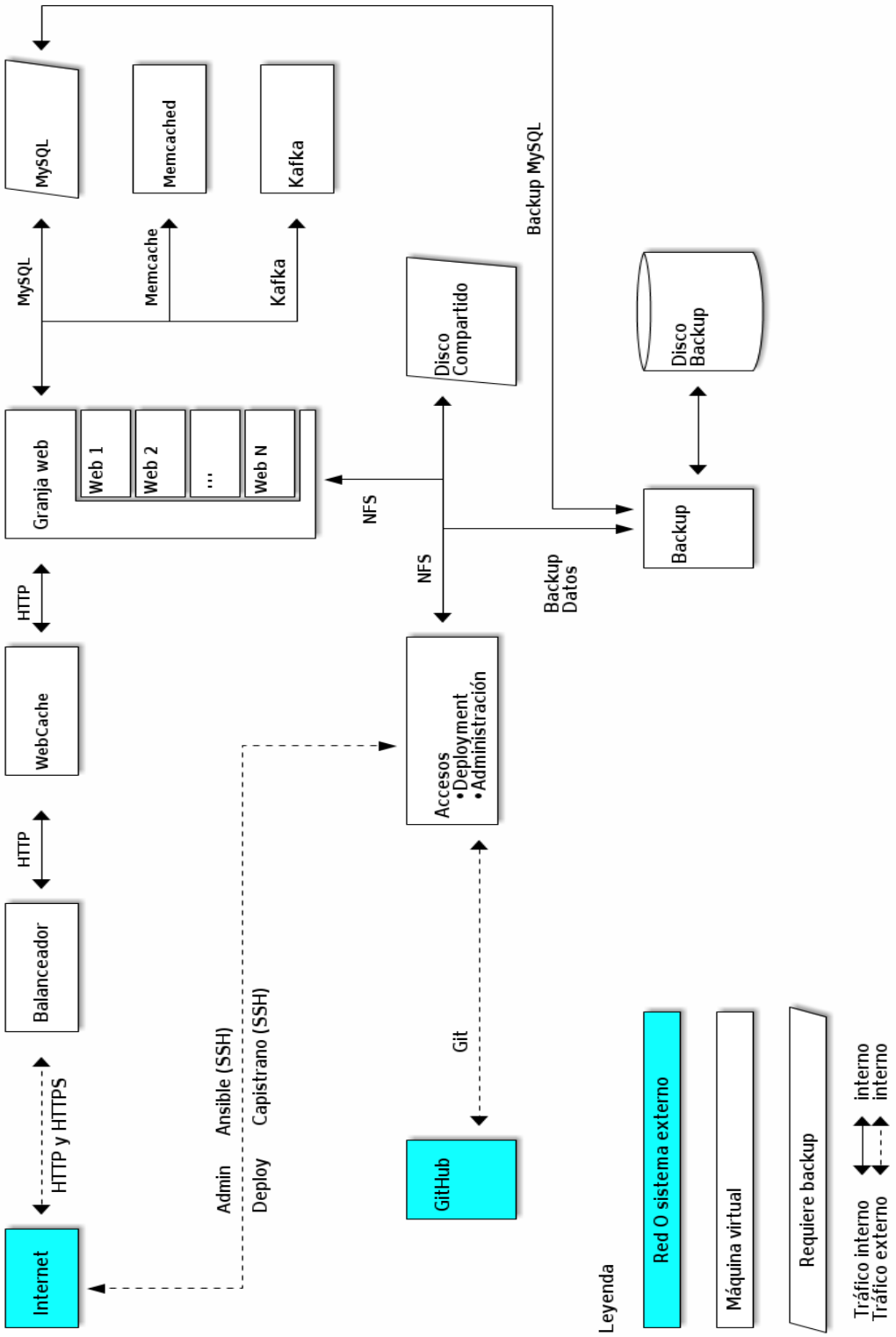

Figura 7.1: arquitectura genérica para Cientópolis

Tráfico interno-interno: hace referencia al tráfico interno de la arquitectura. Aquí se tiene más control sobre los protocolos usados.

Alternativa 1

La Figura 7.2 ilustra la alternativa 1 para la arquitectura de Cientópolis. Como se aprecia, se trata de un diseño en capas que consiste de los siguientes elementos:

Balanceador: se encarga de atender las peticiones HTTPS de los clientes web y derivar las peticiones a la cache web.

Caché: es un servidor que actúa como cache web. Si el contenido pedido por el cliente está en la cache, lo devuelve; si no, envía el requerimiento a los nodos web utilizando una política de *round robin*[1].

Nodos web: se trata de una granja de servidores web[2]. Cada nodo implementa todos los lenguajes soportados por Cientópolis.

Base de datos: se refiere a los motores de bases de datos utilizados por Cientópolis. Al momento de escribir estas páginas se usa solamente MySQL, pero la arquitectura es lo suficientemente extensible como para soportar otros motores fácilmente.

Base de datos en memoria: se utiliza principalmente para almacenar las sesiones de usuarios web, pero también está disponible para las aplicaciones de los desarrolladores. Al momento de escribir se trata de Memcached[3].

Disco compartido: es un disco compartido por los nodos web. En él se encuentra el código de las aplicaciones y los datos que no se guardan en las bases de datos.

Kafka: es un servidor de *streams* de datos. Los procesos productores insertan datos en la plataforma Kafka y los procesos consumidores los retiran.

[1] https://es.wikipedia.org/wiki/Planificaci%C3%B3n_Round-robin
[2] https://en.wikipedia.org/wiki/Server_farm
[3] https://memcached.org/

Máquina de acceso: es un *host* que sirve a dos propósitos: para los administradores, es el lugar desde el cual administran todos los servicios; para los desarrolladores, es el lugar adonde suben las aplicaciones y adonde se conectan para actualizarlas.

Los tipos de servicios brindados por la arquitectura son aquellos necesarios para que las aplicaciones se ejecuten correctamente, así como sus implementaciones y versiones. En cuanto a la disposición en capas, se trata de un uso habitual del **patrón de capas**. Las otras dos alternativas propuestas hacen un uso similar del patrón y se diferencian de la presente arquitectura en la cantidad y uso que se hace de cada capa.

Alternativa 2

La principal razón de utilizar una cache web es la de agilizar la entrega de contenido al usuario. Para que sea útil, la información mostrada debe tener un alto grado de *cachabilidad*. Es decir, la cantidad de aciertos(*hits*) debe ser mayor que la de desaciertos(*miss*). Esto funciona con portales de noticias, blogs y demás sitios en donde la mayoría de los usuarios buscan el mismo contenido, como la noticia del día o el video más visto.

Sin embargo, para el caso de Cientópolis, se espera un grado bajo de *cachabilidad*, debido a que se trata de aplicaciones web en donde la información que se devuelva estará muy ligada a las acciones previas que haya tomado el usuario. Por esta razón se explora un escenario alternativo, mostrado en la Figura 7.3, donde no se tiene una cache web.

La otra diferencia con respecto a la alternativa primera viene dada por el uso que se hace de los nodos web. En este caso cada nodo web se especializa en una sola tecnología o lenguaje. Se pretende, con esta decisión, disminuir la complejidad de cada nodo, mejorando su desempeño y administrabilidad.

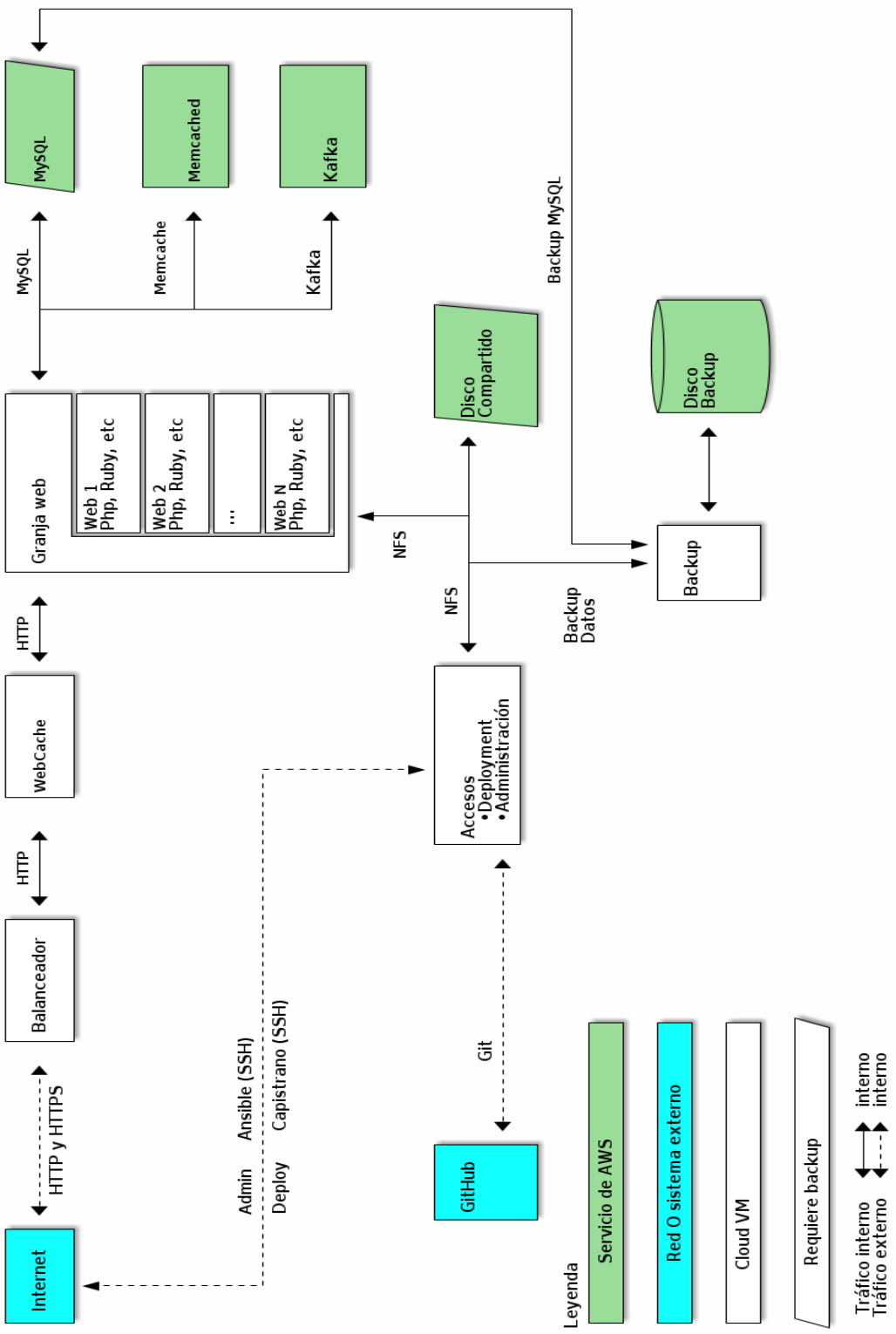

Figura 7.2: alternativa 1 para la arquitectura de Cientópolis

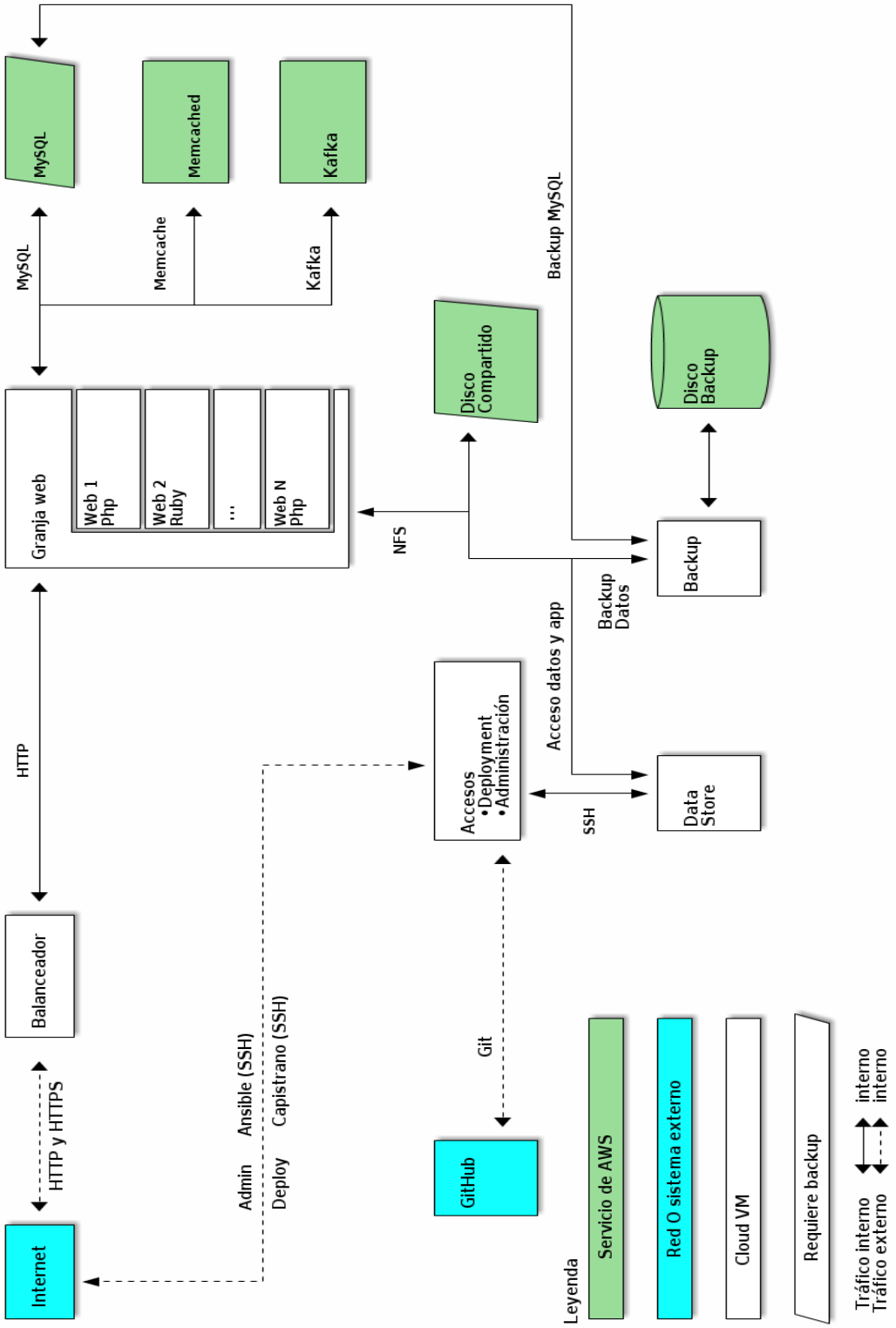

Figura 7.3: alternativa 2 para la arquitectura de Cientópolis

Alternativa 3

La Figura 7.4 muestra la tercera alternativa y se trata, básicamente, de una mezcla entre las dos primeras. Por un lado, tiene todos los elementos de la alternativa 1, incluyendo la cache web; por el otro, los nodos web están especializados, implementan una sola tecnología, en vez de todas, como lo hacen en la alternativa 1.

Además, se espera utilizar ELB (Elastic Load Balancer de Amazon) como balanceador web. Notar que las dos alternativas previas son agnósticas acerca de qué usar como balanceador. Por último, se elige quitar TLS, por lo que el tráfico entrante sería solamente de HTTP.

La justificación para las diferencias que presenta este escenario tiene dos fundamentos: primero, se espera que la mayoría de los usuarios que desplieguen esta arquitectura utilicen Amazon como ambiente de virtualización. Esta decisión puede disminuir la cantidad de organizaciones dispuestas a usar la arquitectura, pero también facilita la implementación de la misma.

En segundo lugar, no se tiene certeza de si es necesario cifrar todo el tráfico entre el cliente web y las aplicaciones de Cientópolis. Después de todo, se trata de ciencia ciudadana con datos públicos. De ahí que se analice la posibilidad de no utilizar HTTPS en la solución final.

7.2. Evaluación de la arquitectura

En esta sección se ejecutarán los pasos del método LAAAM para evaluar las alternativas propuestas y elegir la más apta para el caso de Cientópolis.

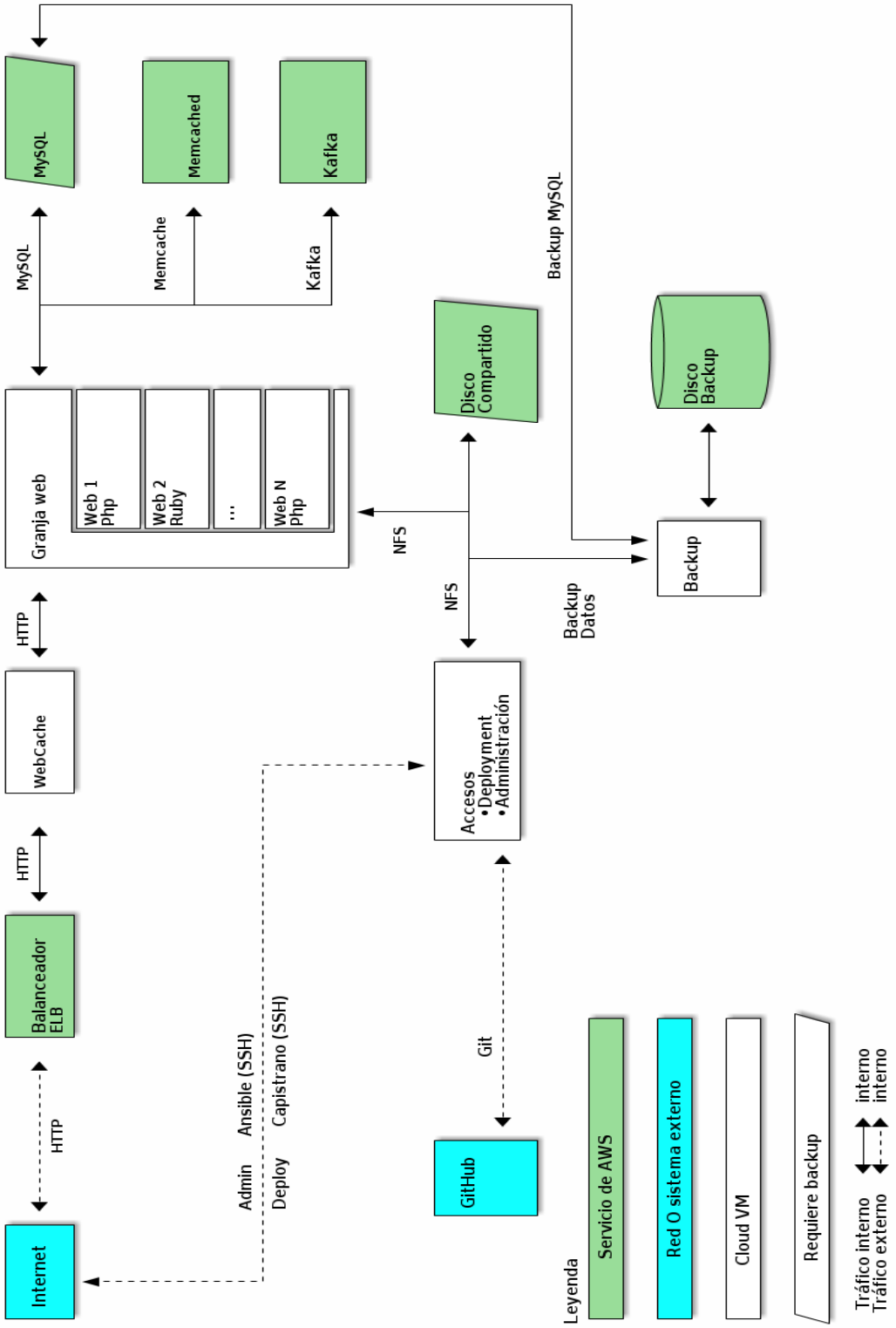

Figura 7.4: Alternativa 3 para la arquitectura de Cientópolis

7.2.1. Procedimiento de evaluación

En esta sección se ejecuta paso a paso el método LAAAM. A modo de resumen, cabe recordar que los dos primeros pasos de LAAAM consisten en identificar los atributos de calidad, es decir, aquellas propiedades que los *stakeholders* ven como las necesidades básicas del sistema; e identificar los escenarios sobre los que se trabajará. LAAAM recomienda realizar un *brainstorming* para detectar tres tipos de escenarios: de casos de uso, de crecimiento y exploratorios. Cuando se presenten los escenarios se indicará cuáles corresponden a casos de uso, cuáles a escenarios de crecimiento y cuáles son exploratorios.

Luego de identificar los atributos de calidad y los escenarios, se procede a priorizar esos escenarios y a asignarles un peso con el fin de generar un *ranking*. Por último, utilizando este *ranking* y las alternativas descritas arriba, se procede a la evaluación y elección de una arquitectura.

Paso 1 - Identificar los atributos de calidad

En la sesión de *brainstorming* llevada a cabo con dos de los *stakeholders* del proyecto Cientópolis se identificaron cuatro atributos de calidad como los más importantes: usabilidad, seguridad, mantenibilidad y performance, en ese orden de prioridad.

El proyecto Cientópolis no dispone de muchos recursos para administrar toda la infraestructura y se espera que quienes instancien la arquitectura por su cuenta, otras facultades o instituciones con sus propios proyectos de ciencia ciudadana, tampoco dispongan de estos recursos. Ésa es la razón por la cual se presenta la usabilidad como el atributo de mayor prioridad. A su vez, se identificaron tres subatributos: instalabilidad, configurabilidad y administrabilidad.

El segundo atributo en importancia es la seguridad. Para los *stakeholders* la seguridad pasa, básicamente, por asegurar medios y conexiones cifradas y por una rápida recuperación ante una falla grave. Esto se manifiesta en la alta puntuación que recibe la confidencialidad y los backups. Los subatributos de seguridad que se identificaron son: confidencialidad, integridad, encriptación (o cifrado), backups, monitoreo y alertas y, por último, auditabilidad.

En tercer lugar aparece la mantenibilidad. Se espera que la nueva arquitectura sea fácilmente mantenible y extensible. Por ejemplo, se desea que agregar soporte para un nuevo lenguaje de programación no sea una tarea *complicada*. Por ejemplo, que lleve menos de una semana de trabajo. Como subatributos se identificaron siete: extensibilidad, adaptabilidad, documentación, testeabilidad, reusabilidad, modularidad y estabilidad.

Notar que los primeros dos atributos, usabilidad y seguridad, tratan con la arquitectura en ejecución; mientras que la mantenibilidad trata con el ciclo de vida de la arquitectura. Y este tercer atributo es de suma importancia para la continuidad de esta arquitectura, dado que su diseño y primera implementación se circunscriben al ámbito de este trabajo de tesis, pero se desea verla evolucionar y acompañar al proyecto Cientópolis a lo largo de todo el ciclo de vida de éste.

El último atributo elicitado es la performance, que cuenta con dos subatributos: escalabilidad y disponibilidad. El fundamento por el cual se considera a la performance en último lugar tiene una explicación sencilla: el objetivo primario es diseñar una arquitectura que sea fácilmente instalable, configurable y administrable y que mantener esa arquitectura no requiera demasiados recursos. En este contexto, la performance pasa a segundo plano.

La Figura 7.5 muestra el *Utility tree* generado a partir de la elicitación de los atributos de calidad. Nótese que al lado de cada atributo se encuentra un número entre paréntesis que indica la prioridad que le corresponde a ese atributo. Los *stakeholders* brindan la prioridad de cada atributo según sus *concerns*.

Pasos 2 y 3 - Identificar los escenarios y asignarles una prioridad

El siguiente paso es identificar los escenarios sobre los que se va a trabajar. Existen tres tipos de escenarios: de casos de uso, de crecimiento y exploratorios. El tipo de escenario se señala con **U**, **C**, **E**, respectivamente. A su vez, los escenarios se caracterizan por tener un contexto, un estímulo y una respuesta. El tipo de escenario es principalmente informativo y sirve para indicar el contexto en el que se lo piensa. Los escenarios calificados como de casos de uso tendrán una prioridad mayor que los de crecimiento; y éstos, a su vez, serán más prioritarios que los exploratorios.

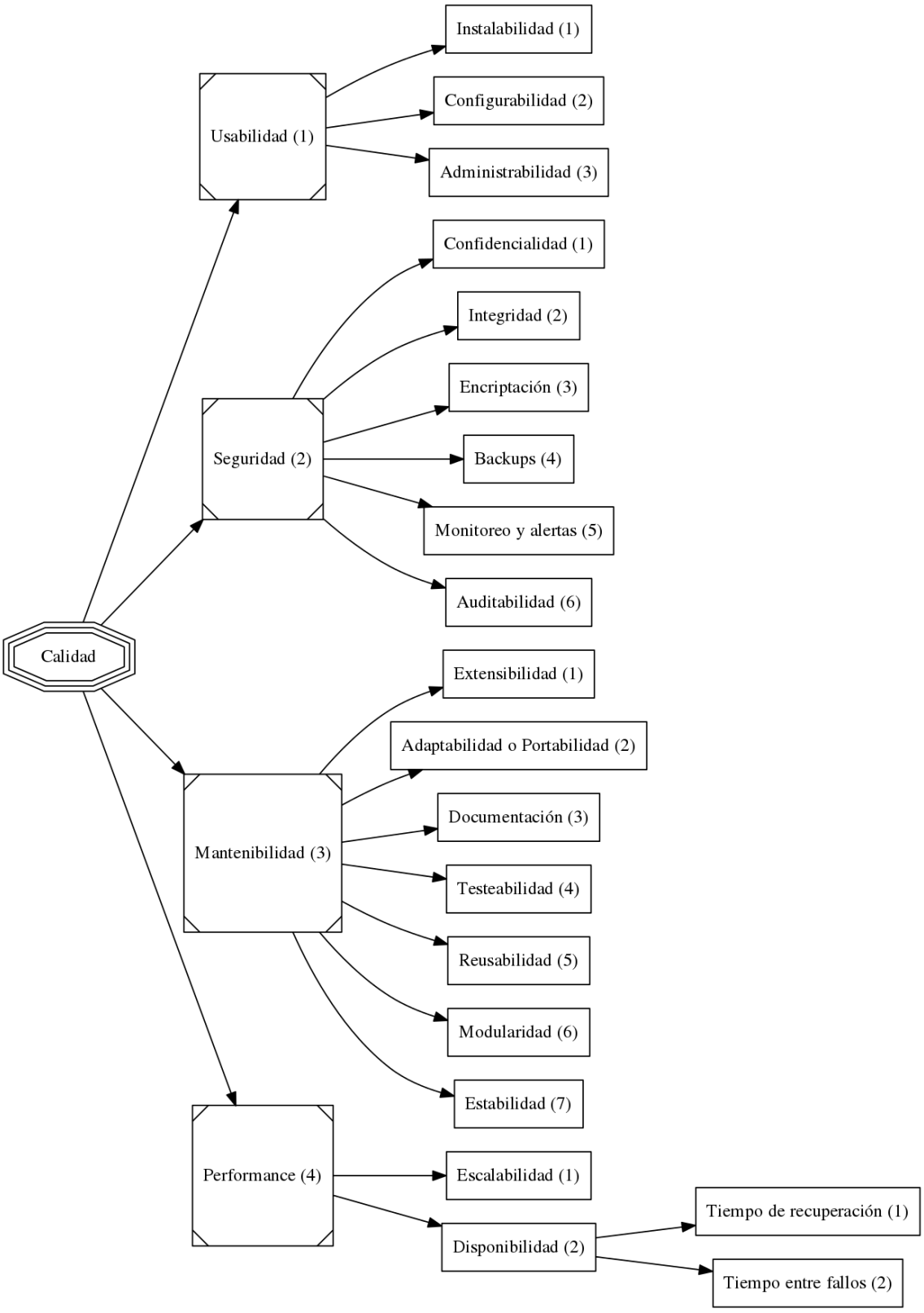

Figura 7.5: *utility tree* que muestra los atributos de calidad identificados.

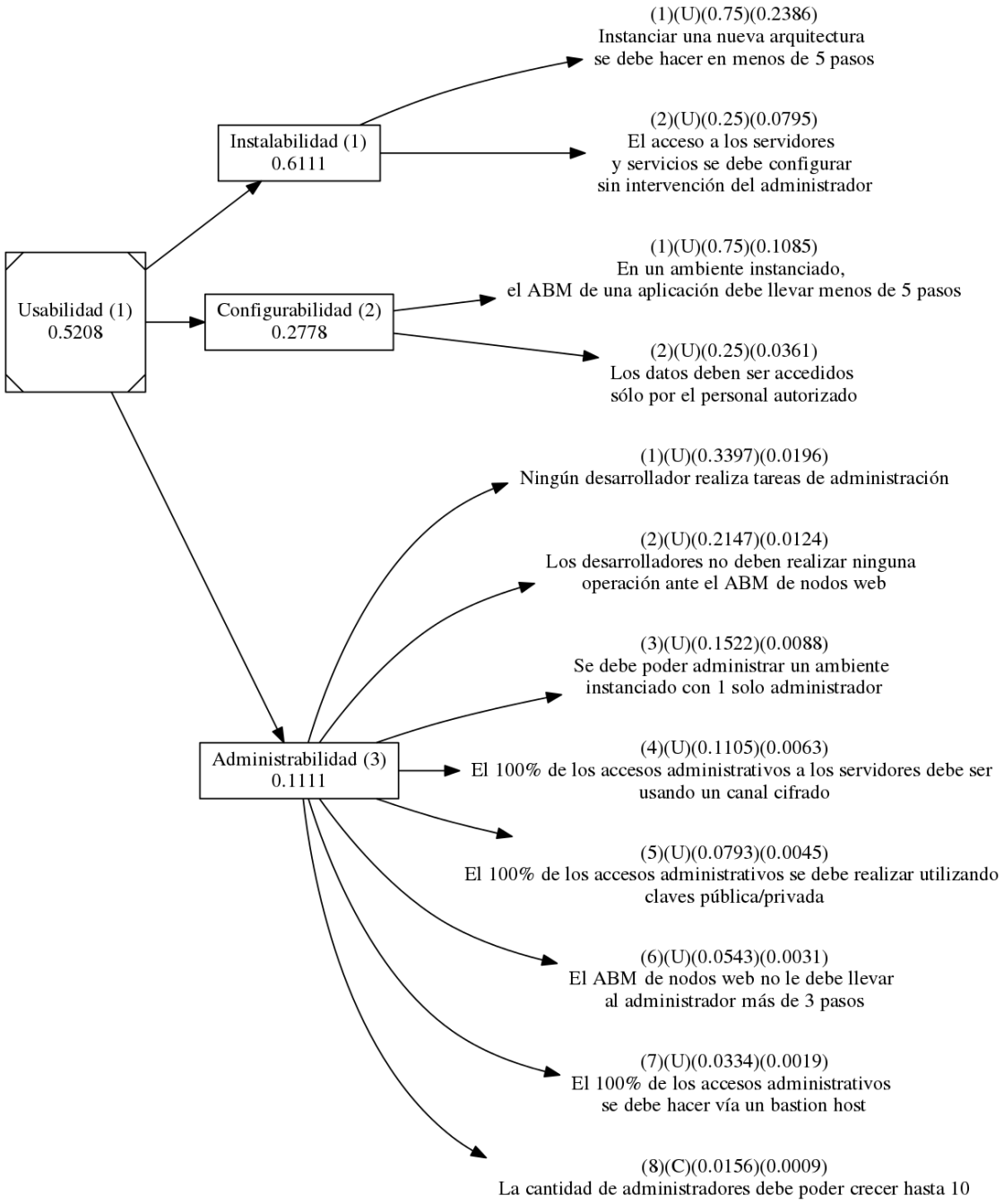

Figura 7.6: *utility tree* que muestra los escenarios para el atributo usabilidad.

Tómese por ejemplo el escenario *usabilidad - instalabilidad - (1)(U)(0.75)(0.2386) Instanciar una nueva arquitectura se debe hacer en menos de 5 pasos*. La primera parte, **Usabilidad**, indica el atributo de calidad. La segunda, **instalabilidad**, es un subatributo de la usabilidad. Luego aparece el escenario en sí precedido de varios números: **(1)(U)(0.75)(0.2386) Instanciar una nueva** [**..**]. El primer número, **(1)**, indica la prioridad relativa; el segundo valor, **(U)** indica que se trata de un escenario de caso de uso; el tercer valor es el peso relativo[4], **(0.75)**, en este caso. Por último, el valor **(0.2386)** es el peso absoluto[5]. Notar que el peso se calcula en el siguiente paso, así que por el momento el lector debe aceptar que esos son los pesos relativo y absoluto correspondientes.

El Cuadro 7.1 muestra todos los escenarios elicitados para el atributo usabilidad. A cada escenario se le asigna un número de escenario (la primera columna), cuya única función es identificar ese escenario, es decir, no asigna ningún tipo de prioridad. Además, se muestra a qué atributo de calidad corresponde. Se indica la prioridad de cada escenario así como su tipo. Por último, se muestra el peso, que se calculará en el paso siguiente.

La Figura 7.6 contiene un *Utility tree* completo (con escenarios) para el atributo usabilidad, que muestra de forma gráfica los datos tabulados en el Cuadro 7.1.

En esta sección se explicará la metodología sólo con el atributo usabilidad. Todos los restantes escenarios se encuentran tabulados en el Anexo-B.

Paso 4 - Asignar un peso a los escenarios

Luego de elicitar y priorizar los escenarios, el siguiente paso es asignarle un peso a cada uno. El Cuadro-7.1 muestra los escenarios de usabilidad con el peso asignado.

[4]El peso con respecto a los demás atributos planteados para **usabilidad - instalabilidad**
[5]El peso con respecto a todos los demás escenarios planteados.

Núm.	Atributo	Prioridad	Tipo	Peso	Escenario
1	Instalab.	1	U	0.2386	Instanciar una nueva arquitectura se debe hacer en menos de 5 pasos.
2	Instalab.	2	U	0.0795	El acceso a los servidores y servicios se debe configurar sin intervención del administrador.
3	Configurab.	1	U	0.1085	En un ambiente instanciado, el ABM de una aplicación debe llevar menos de 5 pasos.
4	Configurab.	2	U	0.0361	Los datos deben ser accedidos sólo por el personal autorizado.
5	Administrab.	1	U	0.0196	Ningún desarrollador realiza tareas de administración.
6	Administrab.	2	U	0.0124	Los desarrolladores no deben realizar ninguna operación ante el ABM de nodos web.
7	Administrab.	3	U	0.0088	Se debe poder administrar un ambiente instanciado con 1 solo administrador.
8	Administrab.	4	U	0.0063	El 100 % de los accesos administrativos a los servidores debe ser usando un canal cifrado.
9	Administrab.	5	U	0.0045	El 100 % de los accesos administrativos se debe realizar utilizando claves pública/privada.
10	Administrab.	6	U	0.0031	El ABM de nodos web no le debe llevar al administrador más de 3 pasos.
11	Administrab.	7	U	0.0019	El 100 % de los accesos administrativos se debe hacer vía un bastion host.
12	Administrab.	8	C	0.0009	La cantidad de administradores debe poder crecer hasta 10.

Cuadro 7.1: escenarios de usabilidad elicitados para Cientópolis.

El procedimiento para asignar pesos es agnóstico a la fórmula usada. Para este trabajo se utiliza una fórmula llamada *Rank Order Centroid* (ROC). La fórmula fue propuesta por Barron y Barrett en su artículo de 1996 Decision Quality Using Ranked Attribute Weights (Barron, F. & Barret, B., 1996). ROC se utiliza para modelar la lógica humana que hay detrás de la toma de decisiones, convirtiendo valores subjetivos ("bien", "muy bien", "mal") en valores analizables matemáticamente. Ver, por ejemplo, el Cuadro 7.2. La expresión general de la fórmula es la siguiente:

$$\frac{\sum_{i=k}^{N} \frac{1}{i}}{N}$$

Donde N es la cantidad de opciones y k es la posición o *ranking*. Por ejemplo, la **usabilidad** tiene un peso de **0.5208** porque está primera de entre cuatro opciones, quedando:

$$\frac{\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4}}{4} = 0{,}5208333333...$$

A su vez, la **instalabilidad** tiene un peso de **0.6111** porque está primera de entre tres opciones, quedando:

$$\frac{\frac{1}{1} + \frac{1}{2} + \frac{1}{3}}{3} = 0{,}611111111...$$

Por último, el escenario 1 del Cuadro 7.1 tiene un peso de **0.75** porque:

$$\frac{\frac{1}{1} + \frac{1}{2}}{2} = 0{,}75$$

Por lo tanto, el peso de ese escenario será: **0.5208** x **0.6111** x **0.75** = 0.2386[6]. Como puede verse, este paso es bastante automático. En el caso del escenario 2 el peso es de **0.25**:

$$\frac{\frac{1}{2}}{2} = 0{,}25$$

[6]Se toma una aproximación de cuatro dígitos decimales. Para este caso el número completo se aproxima a 0.23869566...

Usabilidad				
Núm.	Peso	Alternativa 1	Alternativa 2	Alternativa 3
1	0.2386	Muy Bien	Muy Bien	Muy Bien
2	0.0795	Muy Bien	Muy Bien	Muy Bien
3	0.1085	Muy Bien	Muy Bien	Muy Bien
4	0.0361	Muy Bien	Muy Bien	Muy Bien
5	0.0196	Muy Bien	Muy Bien	Muy Bien
6	0.0124	Bien	Muy Bien	Bien
7	0.0088	Muy Bien	Muy Bien	Bien
8	0.0063	Muy Bien	Muy Bien	Muy Bien
9	0.0045	Muy Bien	Muy Bien	Muy Bien
10	0.0031	Muy Bien	Muy Bien	Muy Bien
11	0.0019	Muy Bien	Muy Bien	Bien
12	0.0009	Muy Bien	Muy Bien	Muy Bien
Total	0.5202	1.0279	1.0404	1.0172

Cuadro 7.2: escenarios de usabilidad y cómo se ajusta cada alternativa.

En el Anexo B están las tablas de escenarios con los pesos calculados. Como se dijo anteriormente, aquí se elige escribir un breve ejemplo para que el lector entienda la metodología y se lo invita a revisar dicho anexo para revisar los datos completos.

Paso 5 - Evaluación de las alternativas

El último paso es comparar cómo se ajustan las alternativas de arquitectura a cada escenario. El Cuadro 7.2 muestra los cálculos para el atributo usabilidad. Para cada alternativa se indica qué tan bien se satisface el escenario específico. Los valores elegidos son: **Mal (0)**, **Bien (1)** y **Muy Bien (2)**.

Como muestra el Cuadro 7.2, la alternativa 2 exhibe una mayor puntuación que las demás al momento de satisfacer los escenarios de usabilidad. En el Anexo B se encuentran los cálculos completos, de donde se deduce que es la Alternativa 2 la que mejor satisface la mayoría de los escenarios.

7.3. Propiedades de la arquitectura propuesta

En la sección previa se evaluaron tres alternativas para la arquitectura de Cientópolis utilizando el método LAAAM y se mostró que la *mejor* arquitectura era la llamada Alternativa 2. En esta sección se hablará sobre las propiedades presentes en la arquitectura propuesta.

Usabilidad: como se mencionó, es complicado medir la usabilidad. Sin embargo, se espera que con el uso de herramientas de automatización la instanciación y configuración de la arquitectura mejore su usabilidad.

Performance: se espera que el desempeño o performance de la arquitectura se vea incrementado al utilizar diferentes servidores, para aplicaciones y para bases de datos. También se espera que al negociar la sesión TLS en un servidor separado se mejore la performance general del sistema. Por último, al utilizar herramientas de administración automatizadas se mejora también la performance al momento de administrar la arquitectura.

Instalabilidad: como ocurre con la usabilidad, el uso de herramientas de automatización supone una mejora en lo relacionado con la instalabilidad de la arquitectura. Además, las herramientas de monitoreo provistas también ayudan en este sentido, dado que muestran que todos los servicios funcionan correctamente.

Mantenibilidad: el uso de herramientas de gestión automatizadas y la disposición de un ambiente de pruebas para el desarrollo de la arquitectura aseguran un mejor grado de mantenibilidad de la arquitectura. De igual forma, la separación de funcionalidad en diferentes nodos también mejora esta propiedad.

Documentación: en cuanto a este ítem, el código que instancia y configura la arquitectura funciona como documentación, de forma que se minimiza la documentación que se debe mantener.

Adaptabilidad o portabilidad: la identificación de todas las tecnologías utilizadas así como su separación en diferentes nodos mejora la portabilidad de la arquitectura. Sin embargo, también se debe remarcar que portar la arquitectura hacia otra solución de *cloud* (o a servidores físicos) requeriría de la reimplementación de varias lineas de código.

Auditabilidad: la arquitectura planteada no mejora en nada la auditabilidad del sistema. Existirán más registros o *logs*. Por ejemplo, cada usuario tendrá su propio registro de autenticación. Sin embargo, no se proveen herramientas de AAA (Authentication, Authorization, and Accounting) como puede ser Radius[7]. Tampoco se brinda una solución de archivo o *archiving*. De todas maneras, sería relativamente sencillo mejorar esta propiedad si algún *stakeholder* lo requiere.

Configurabilidad: nuevamente la decisión de automatizar la arquitectura supone una mejora en este ítem.

Extensibilidad: al utilizar la nube de Amazon se cuenta con un gran número de servicios que se pueden utilizar sin necesidad de instalarlos uno mismo, lo cual mejora la extensibilidad del sistema. Por otra parte, las instancias ejecutando en otras nubes deberán antes implementar las extensiones necesarias, como puede ser agregar compatibilidad con LDAP. Sin embargo, esta tarea también se ve aliviada debido a la mejora en la mantenibilidad en general. Por último, para el caso de Proxmox, los servicios similares a los provistos por AWS se implementan con diversas aplicaciones open source.

Escalabilidad: la separación en diferentes nodos para cada servicio facilita la escalación del sistema. Es posible escalar el servidor de bases de datos mientras se dejan intactos los nodos web. El diseño también soporta el incremento horizontal de los nodos web.

Administrabilidad: las prácticas de infraestructura como código mejoran en general la administrabilidad de los sistemas.

[7] https://en.wikipedia.org/wiki/RADIUS

Modularidad: comparando la arquitectura previa con la aquí propuesta se nota un alto grado de modularidad. Incluso entre las alternativas planteadas también hay diversos grados de modularidad. Por ejemplo, la Alternativa 1 es *menos* modular que la Alternativa 2, dado que en la primera, todas las tecnologías web conviven en todos los nodos, mientras que en la segunda, un nodo implementa una sola tecnología web.

Reusabilidad: la infraestructura como código se encuentra versionada y es relativamente sencillo construir una nueva arquitectura tomando la primera como base.

Seguridad: el uso de TLS para todas las aplicaciones supone una mejora considerable frente al panorama previo, donde no existía un esquema claro sobre si una aplicación debía correr sobre HTTPS o no. Con el nuevo diseño, todas las aplicaciones están obligadas a correr sobre HTTPS[8]. También, el hecho de que cada usuario se conecta utilizando su propio nombre de usuario y no uno genérico, como venía ocurriendo, mejora notablemente la seguridad del sistema.

Estabilidad: se espera haber capturado la mayoría de los requerimientos y expectativas de los *stakeholders*, razón por la cual al finalizar este trabajo debería ser baja la cantidad de cambios.

Testeabilidad: usando tests automatizados sobre la arquitectura de prueba se mejora esta propiedad. Como se verá más adelante, esta implementación queda como trabajo a futuro.

7.4. Resumen

En este capítulo se mostraron tres alternativas para la nueva arquitectura de Cientópolis y se utilizó el método LAAAM para elegir la más apropiada según las necesidades de los *stakeholders* del proyecto. Además, se enumeraron las propiedades que exhibirá la nueva arquitectura una vez implementada.

[8] Algunas aplicaciones podrían dejar de funcionar correctamente en caso de usar sólo HTTPS o el navegador web podría mostrar alguna advertencia si en la aplicación se mezcla HTTP con HTTPS. Si bien esto es un problema, es de fácil resolución.

El siguiente capítulo trata sobre las arquitecturas programables y se presentan las herramientas utilizadas para la implementación de la arquitectura aquí propuesta.

Capítulo 8

Propuesta de arquitectura programable

> Siempre deseé que mi computadora fuera tan fácil de usar como mi teléfono. Mi deseo se ha hecho realidad: ya no sé usar mi teléfono.
>
> *Bjarne Stroustrup*

Antes de avanzar con la implementación de la arquitectura es necesario mencionar y explicar varias de las herramientas que se utilizarán. Este capítulo trata de las tecnologías usadas en la implementación y gestión de la arquitectura.

8.1. Devops como nuevo paradigma de administración de infraestructura

Devops surge de la interdependencia entre los desarrolladores y los administradores de sistemas. Es un perfil intermedio o híbrido, que aparece de la mezcla de conocimiento como alternativa al trabajo en silos aislados (Ebert, C., Gallardo, G., Hernantes, J. & Serrano, N., 2016). En los últimos años se han desarrollado una gran variedad de herramientas para asistir a las tarea de los devops, como son Ansible, Terraform o Docker, entre muchas otras.

En el capítulo introductorio de este trabajo se habla acerca de cómo las metodologías ágiles presionaban a los administradores para instanciar ambientes de forma cada vez más rápida y se indicaba que *la respuesta dada por la Informática fue la sistematización y automatización del despliegue de aplicaciones y de la configuración de los servidores*. Este tipo de automatización se da utilizando herramientas de devops.

En este trabajo se han utilizado algunas de esas herramientas pero en ningún momento se ha pretendido instaurar la semilla del movimiento en el grupo de desarrolladores y administradores de Cientópolis. Citando nuevamente el artículo **DevOps** (Ebert et al., 2016), "Devops implica un cambio cultural hacia la colaboración entre los equipos de desarrollo, operaciones y calidad". Ese cambio cultural escapa al alcance de esta tesis y es para un trabajo a futuro.

La empresa Puppet (Creadora de Puppet y pionera en el movimiento de automatización de servidores y devops) realiza desde año el 2014 un reporte anual basado en la encuesta de miles de usuarios alrededor del mundo. En su *State of DevOps Report* (Puppet, 2017) menciona que en 2017 el porcentaje de grupos de devops alcanza el 27 %, habiendo sido del 16 % en 2014, contra un 25 % de desarrolladores y un 28 % de administradores para 2017.

8.2. Amazon, Proxmox y alternativas

Como se mencionó páginas atrás, Amazon[1] dona tiempo de cómputo al proyecto Cientópolis, y es por esta razón que se decide aprovechar la oportunidad para probar los servicios ofrecidos.

Una de las ventajas que hay en trabajar con la nube de Amazon es el hecho de que brinda varios servicios bien conocidos. Por ejemplo, en vez de instalar un servidor de bases de datos MySQL, se va a utilizar el servicio de MySQL brindado por Amazon.

[1] https://aws.amazon.com/es/

Existe una gran variedad de servicios de Amazon para el uso público. Algunos son aprovechados por esta arquitectura mientras que otros no son necesarios o se decidió no incluirlos. Entre estos últimos, el más notorio es ELB, Elastic Load Balancer, que es un servicio para balancear tráfico web. Se decidió no utilizarlo porque volvía más compleja la implementación de la automatización del proxy reverso con TLS utilizando Let's Encrypt.

En la actualidad existe una gran variedad de alternativas a Amazon, como son Digital Ocean[2], Rackspace[3] (cocreadores de OpenStack[4] junto con la NASA), Linode[5], Azure[6] y Google Cloud[7], entre muchísimos otros.

La empresa Torry Harris[8] publicó Cloud Computing Services – A comparison[9], que es una comparación entre las nubes de Amazon AWS, Google App Engine, Windows Azure, Force.com, Racks pace, GoGrid. Se trata de un estudio relativamente viejo (febrero de 2013), pero que sirve para ilustrar la complejidad del mundo de las nubes.

Los servicios ofrecidos por las empresas de *cloud computing* se agrupan típicamente en tres conjuntos: IaaS, PaaS y SaaS. A continuación se describen brevemente estos tres modelos de servicio, de modo que el lector tenga una mejora comprensión de los servicios utilizados en este trabajo.

IaaS - Infrastructure as a service: se trata de infraestructura o *hardware* virtualizado. Incluye cuestiones como direccionamiento de red, *data centers*, ancho de banda, ruteo y *firewalls*, entre otros.

PaaS - Platform as a service: los servicios de PaaS están orientados principalmente a los desarrolladores, a quienes se les brinda una plataforma (Sistema operativo) junto con un entorno de programación.

---

[2]

[3] https://www.rackspace.com/es-ar

[4] https://www.rackspace.com/es-ar/openstack

[5] https://www.linode.com/

[6] https://azure.microsoft.com/es-es/

[7] https://cloud.google.com/?hl=es

[8] http://www.thbs.com/

[9] https://www.thbs.com/thbs-insights/comparison-of-cloud-computing-services

SaaS - Software as a service: son aplicaciones a las cuales los usuarios pueden acceder directamente a través de Internet.

Como ocurre habitualmente, es muy común que las herramientas presenten características de más de un grupo. También la línea divisoria entre qué se considera IaaS y qué PaaS, por ejemplo, es muy fina y subjetiva. Está fuera del alcance de este trabajo profundizar sobre las ventajas y desventajas de los diferentes modelos de servicio, así como realizar una comparación entre los distintos proveedores.

Existen herramientas que permiten construir nubes privadas sobre *hardware* propio. Algunas son multiplataforma y funcionan tanto en Linux como en Windows, Mac y algunos BSDs, mientras que otras están ligadas a un sistema operativo en particular. Las soluciones de virtualización más utilizadas para construir nubes privadas son:

Proxmox: es un entorno de virtualización[10] mediante el cual se pueden crear máquinas virtuales con KVM[11] o contenedores usando LXC[12]. Se trata de una solución basada en Linux que puede virtualizar diversos sistemas operativos (Windows, Linux, FreeBSD, entre otros). Una de las inquietudes presentadas por los *stakeholders* de este proyecto es la compatibilidad de la arquitectura con Proxmox. Es decir, la posibilidad de instanciar la arquitectura de Cientópolis en Proxmox.

VirtualBox: es un producto de virtualización[13] multiplataforma que corre sobre Linux, Windows, Mac y Solaris y al igual que Proxmox, puede virtualizar una gran cantidad de sistemas operativos.

ESXi: la solución de virtualización de VMWare[14] corre sobre su propio sistema operativo (ESXi) y puede virtualizar diversos sistemas operativos. VMWare también ofrece un producto multiplataforma llamado Workstation.

[10] https://www.proxmox.com/en/
[11] https://www.linux-kvm.org/page/Main_Page
[12] https://linuxcontainers.org/
[13] https://www.virtualbox.org/
[14] https://www.vmware.com/ar/products/vsphere-hypervisor.html

XEN: es otro entorno de virtualización originalmente desarrollado en la Universidad de Cambridge[15]. La empresa Citrix Systems presenta una versión -también *open source*- llamada XenServer[16].

Hyper-V: es la solución de Microsoft[17]. Obviamente corre sólo sobre Windows pero puede virtualizar otros sistemas opertivos aparte de Windows[18].

8.2.1. Acerca de los contenedores

Puede pensarse en los contenedores (*containers* en inglés) como ambientes de ejecución auto-contenidos. En el caso de las máquinas virtuales, el virtualizador se encarga de virtualizar distintos elementos de *hardware*, como CPU, memoria, disco, buses, interfaces físicas de red, BIOS, etc. Mientras que en el caso de los contenedores, el sistema operativo *virtualiza* elementos del sistema operativo, como la estructura de directorios, los usuarios del sistema o el árbol de procesos.

Si bien la idea detrás de los contenedores es antigua, en los últimos años han empezado a gozar de gran popularidad, principalmente en el ámbito del desarrollo de aplicaciones web. Habitualmente se utilizan contenedores para empaquetar una aplicación y todas sus dependencias, de modo que se facilita su distribución e instalación. Por ejemplo, un contenedor puede incluir una aplicación web, las librerías que requiere, y el servidor web para acceder a esa aplicación. Al momento de escribir estas páginas, el producto de *contenedorización* más usado es Docker[19].

Sin embargo, al dar soporte para contenedores se pierde el control de la arquitectura, porque se delegan decisiones de arquitectura a los desarrolladores[20]. Para utilizar contenedores de forma segura y consistente, se debería brindar a los desarrolladores toda la infraestructura para crear, probar y desplegar los contenedores de forma segura, lo cual es motivo para otro trabajo.

[15] https://www.xenproject.org/

[16] https://xenserver.org/

[17] https://technet.microsoft.com/en-us/library/mt169373

[18] https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/supported-linux-and-freebsd-virtual-machines-for-hyper-v-on-windows

[19] https://www.docker.com/

[20] En realidad se delega en quienes armen los contenedores, pero en el caso particular de Cientópolis, terminarían siendo los desarrolladores

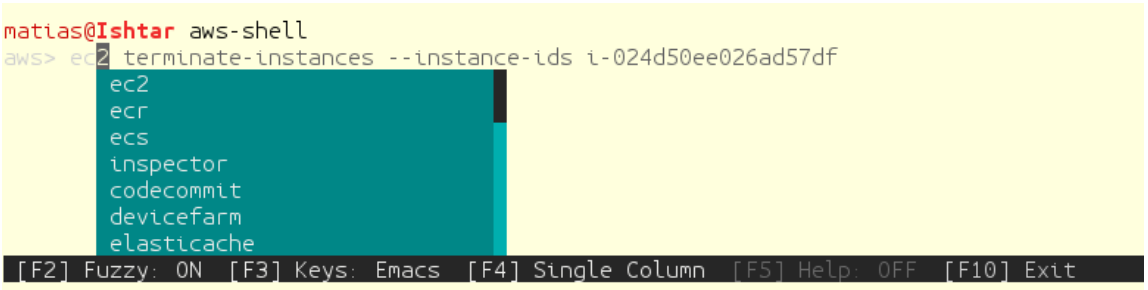

Figura 8.1: AWS-Shell en ejecución. Notar la facilidad de autocompletado provista

Uno de los puntos más cuestionados acerca de los contenedores es el nivel de aislamiento que implementan y qué tan posible es para un proceso que corre dentro de un contenedor *salir* de él y acceder a otros contenedores o al sistema anfitrión. Desde la versión 1.10 (febrero de 2016), Docker soporta el espacio de nombres para los usuarios del sistema[21], lo cual permite mapear usuarios privilegiados adentro del contenedor a usuarios no privilegiados en el sistema anfitrión.

Otra cuestión a tener en cuenta es que los contenedores están atados a un sistema operativo en particular. Utilizar contenedores para alojar una aplicación implica que esa aplicación estará atada a un sistema operativo determinado.

Por último, para cerrar el tema de los contenedores, hay que mencionar que Docker no es la única tecnología de contenedores; existen otras, como LXC[22] u OpenVZ[23].

8.2.2. Sobre el uso de AWS

Existen dos formas de utilizar los servicios web de Amazon (AWS, *Amazon Web Services*): vía web o usando la línea de comandos (*shell*). Para este trabajo se utilizó la segunda opción. Sin embargo, antes de pasar a describir la CLI (siglas de *Command-line interface* en inglés) de AWS, es preciso mencionar por qué se descartó la opción web.

[21] https://docs.docker.com/engine/security/userns-remap/ y http://man7.org/linux/man-pages/man7/user_namespaces.7.html
[22] https://linuxcontainers.org/
[23] https://openvz.org/Main_Page

La administración a través de la web podría parecer más cómoda e intuitiva, pero la imposibilidad inherente de automatización lleva a dos problemas graves. Por un lado, la administración a mano de la arquitectura se tornaría tediosa y no escalaría. Por otro, el proceso de instanciar otra vez la misma arquitectura debería realizarse manualmente, por lo que es imposible garantizar que la nueva instanciación presentará las mismas características que la previa. Como se dijo anteriormente, la única manera de garantizar un determinado grado de calidad y repetibilidad en un proceso es mediante su automatización.

La **AWS Command Line Interface (CLI)**[24] es una herramienta que sirve para manejar todos los servicios de la nube de Amazon. Es posible utilizar esta herramienta para escribir *scripts* de automatización.

Por ejemplo, el comando **aws ec2 describe-instances** sirve para listar todas las instancias (máquinas virtuales) en AWS, estén o no prendidas. El *output* de ese comando es:

Listing 1: salida del comando **aws ec2 describe-instances**

```
1
2   "Reservations": [
3       {
4           "OwnerId": "352893117079",
5           "ReservationId": "r-0b84a1aaa6b24eda1",
6           "Groups": [],
7           "Instances": [
8               {
9                   "Monitoring": {
10                      "State": "disabled"
11                  },
12                  "PublicDnsName": "",
13                  "RootDeviceType": "ebs",
14                  "State": {
15                      "Code": 16,
```

[24] https://aws.amazon.com/es/cli/ y https://github.com/awslabs/aws-shell

16 . . . . . .

Como se ve, las respuestas a los comandos están codificadas en JSON (JavaScript Object Notation)[25].

La CLI también funciona en modalidad interactiva: ejecutando el comando **aws-shell** se obtiene un *prompt* como el que se muestra en la Figura-8.1. El modo interactivo provee de una funcionalidad de autocompletado muy útil que, incluso, sirve para autocompletar los IDs de los recursos en AWS, como IDs de *security groups*, instancias, etc.

Sin embargo, la CLI es una alternativa poco amigable para los usuarios que no están familiarizados con la linea de comandos. Más aún, se espera que las personas que instancien su propia arquitectura de Cientópolis no dispongan necesariamente de las habilidades para trabajar con la linea de comandos. Es por esta razón que se requiere alguna herramienta de más alto nivel.

8.2.3. Sobre el uso de Proxmox

Los servicios ofrecidos por Proxmox no son tan amplios como los disponibles en AWS. Por ejemplo, mientras que AWS provee motores de bases de datos, con Proxmox se debe instalar *a mano* el motor en una máquina virtual. Como resultado, la solución basada en Proxmox deberá implementar más servicios, mientras que la basada en AWS podrá consumir directamente esos servicios desde Amazon.

Tal como ocurre con AWS, Proxmox permite gestionar los recursos a través de una interfaz web o a través de la linea de comandos. En la sección anterior se explicó las ventajas de utilizar AWS por linea de comandos en vez de la interfaz web y lo mismo aplica para Proxmox.

8.2.4. Sobre las herramientas elegidas

Habiendo tantas herramientas para trabajar, caben dos preguntas importantes:

- ¿Por qué se eligieron Proxmox y AWS por sobre las demás alternativas?

[25] https://json.org/

- ¿Por qué se usaron máquinas virtuales en vez de contenedores?

La elección de Proxmox y AWS viene dada por el contexto. Como se explicó anteriormente, la empresa Amazon dedicó recursos académicos que se terminaron usando en el proyecto Cientópolis; por otro lado, el laboratorio cuenta con virtualizadores Proxmox adonde residirían las aplicaciones una vez se agoten los recursos brindados por AWS. Por lo tanto, no existía la posibilidad de plantear el uso de otras tecnologías, como XenServer o VMware.

En cuanto a la segunda pregunta, y como se comentó anteriormente, el uso de Docker hubiera implicado una pérdida de control sobre la arquitectura final. También hubiera implicado agregar una nueva tecnología al proyecto, complejizando la solución final.

8.3. Ansible y alternativas

Los procesos exhiben un mayor grado de calidad cuando son repetibles y, si bien una persona puede ejecutar reiteradas veces el mismo proceso, suele hacerlo de forma lenta y es propenso al error. Por otro lado, la automatización de dichos procesos mejora los tiempos de ejecución y permite hablar de un proceso 100 % repetible. Es, mediante esta repetibilidad asegurada por la automatización lo que permite asegurar que todos los ambientes de Cientópolis serán iguales. Es decir, se conservan las propiedades de calidad a través de las instanciaciones.

8.3.1. ¿Por qué utilizar alguna de estas herramientas?

La CLI de AWS se utilizó para realizar pruebas y verificar que los recursos manejados con Ansible respeten las especificaciones dadas. Sin embargo, como se aclaró, dicha CLI es incómoda para los usuarios menos experimentados con la linea de comandos, la clase de usuarios que se espera creen nuevas instancias de la arquitectura de Cientópolis.

Más aún, se le recuerda al lector tres requerimientos importantes de usabilidad manifestados por los *stakeholders*:

- Instanciar una nueva arquitectura se debe hacer en menos de 5 pasos.

- En un ambiente instanciado, el ABM[26] de una aplicación debe llevar menos de 5 pasos.
- El ABM de nodos web no le debe llevar al administrador más de 3 pasos.

De lo mencionado más arriba surge la necesidad de automatizar las tareas relacionadas con la creación y mantenimiento de la arquitectura. La herramienta elegida para automatizar dichas tareas es Ansible.

8.3.2. ¿Por qué se eligió Ansible?

Desde hace más de cinco años en el área de IT se tiende a utilizar herramientas de automatización para la instanciación y configuración de diferentes servicios. Ejemplos de estas herramientas son Chef[27], Ansible[28], Salt[29] y Puppet[30], esta última vista como precursora del movimiento. ¿Por qué se eligió Ansible y no cualquiera de las otras?

La respuesta tiene varias partes. Por un lado, el desarrollador de la arquitectura ya estaba familiarizado con Ansible, lo cual fue visto como una gran ventaja con respecto a las otras herramientas.

En A Study of Configuration Management Systems, Solutions for Deployment and Configuration of Software in a Cloud Environment (Torberntsson, K. & Rydin, Y., 2014) se analizan las principales herramientas de automatización, incluyendo las cuatro aquí mencionadas. La Tabla-1 de dicho informe (En la página 8), muestra una comparación exhaustiva de todas las herramientas. Cabe aclarar que, siendo el informe del junio de 2014, es incorrecto acerca del soporte de Ansible para Windows. Según el informe, Ansible no tiene soporte para Windows, pero al momento de escribir estas líneas ya lo tiene incorporado[31] (de hecho lo incorpora en Agosto de 2014, en su versión 1.7).

En el Cuadro 8.1 se reproduce parte de dicha tabla, actualizada, mencionando sólo las herramientas Ansible, Chef, Puppet y Salt.

[26] Siglas de Alta, Baja y Modificación.
[27] https://www.chef.io/
[28] https://www.ansible.com/
[29] https://saltstack.com/
[30] https://puppet.com/
[31] http://docs.ansible.com/ansible/intro_windows.html

	Comunidad activa	Soporte comercial	Actualización	Soporte para Windows	Soporte para MacOSX
Ansible	Sí	Sí	Julio 2018 (2.6.1)	Sí	Sí
Chef	Sí	Sí	Febrero 2018 (12.17.33)	Sí	Sí
Puppet	Sí	Sí	Julio 2018 (5.5.3)	Sí	Sí
Salt	Sí	Sí	Junio 2018 (2018.3.2)	Sí	Sí

Cuadro 8.1: cuadro comparativo de las herramientas Ansible, Chef, Puppet y Salt.

Además, Ansible presenta una curva de aprendizaje más plana, por lo que se espera que la transferencia de conocimiento sea más rápida y fácil. Como se menciona en A Study of Configuration Management Systems (Torberntsson, K. & Rydin, Y., 2014): *"Quizás la cualidad que mejor define a Ansible es su simplicidad. Michael DeHaan comenzó el desarrollo de Ansible como una reacción a Puppet, el cual había usado pero pensaba que era demasiado complejo"*.

Por otra parte, Paul Venezia escribió una comparación entre Puppet, Chef, Ansible y Salt que fue publicada en ComputerWorld (Venezia, P., 2013), señalando que la audiencia a la que están destinadas estas herramientas es diferente. Puppet y Chef está más cerca del mundo de los desarrolladores, mientras que Ansible y Salt está más cercana al de los administradores.

En cuanto a lo puramente técnico, Ansible no requiere (y, de hecho, no dispone) de un agente en el lado del servidor. Tampoco depende de un servidor central, por lo que instalar y mantener una solución basada en Ansible es mucho más económico en términos de tiempo y esfuerzo.

Además, los módulos de Ansible abarcan la mayoría de las tareas que se realizan con regularidad en los servidores, haciendo de Ansible una herramienta sumamente expresiva. Ansible utiliza el paradigma de *programación declarativa* por medio del lenguaje YAML (YAML Ain't Markup Language)[32].

[32] http://yaml.org/

Como última cuestión técnica se debe mencionar que Ansible y Salt utilizan el modelo *push*, en el cual el administrador *empuja* la configuración hacia los servidores, mientras que Chef y Puppet utilizan el modelo *pull*, es decir, los servidores *buscan* la configuración en un servidor central. Para el contexto de este trabajo, estas diferencias no aportan demasiado y se mencionan solamente por completitud.

Acerca de la continuidad del proyecto, Ansible fue comprado por Red Hat en 2015 y es la herramienta para la gestión de infraestructura más marcada en GitHub (Geerling, J., 2018). Aquí se traduce como *marcado* el concepto de señalar con la estrella un proyecto de GitHub[33].

8.3.3. Breve introducción a Ansible

Ansible[34] es una herramienta de software libre que permite automatizar la configuración de servidores y el *deployment* de aplicaciones. La primera versión apareció en 2012 y actualmente pertenece a la empresa Red Hat. Cuenta con un repositorio de la comunidad llamado Galaxy[35], donde se pueden encontrar miles de *playbooks*. Los *playbooks* sirven para definir los pasos que componen un procedimiento determinado. Ansible llama a cada uno de estos pasos *tasks* o tareas.

Otra componente importante de Ansible son los módulos, que sirven para controlar distintos recursos. Por ejemplo, el módulo **user**[36] sirve para manejar los usuarios de un sistema.

Por otra parte, el "inventario" o *inventory* en inglés es el listado de servidores sobre los cuales se va a realizar alguna tarea administrativa. Como dice en la documentación oficial de Ansible, "Si los módulos de Ansible son las herramientas en tu taller, los *playbooks* son tus manuales, y el inventario de servidores es tu materia prima"[37].

A continuación se muestra un *playbook* de ejemplo que aparece en el sitio de Ansible:

[33] https://help.github.com/articles/about-stars/
[34] https://www.ansible.com/
[35] https://galaxy.ansible.com/
[36] http://docs.ansible.com/ansible/user_module.html
[37] https://docs.ansible.com/ansible/latest/user_guide/playbooks.html

Listing 2: ejemplo de *playbook*

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum: name=httpd state=latest
    - name: write the apache config file
      template: src=/srv/httpd.j2 dest=/etc/httpd.conf
      notify:
        - restart apache
    - name: ensure apache is running (and enable it at boot)
      service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

Lo primero que se aprecia es que el lenguaje usado por Ansible es YAML[38]. De hecho, Ansible utiliza dos lenguajes solamente: Jinja2[39] para los *templates* y YAML para el resto. A continuación se describe brevemente cada ítem del *playbook*:

hosts: hace referencia a un grupo de hosts del inventario. En este caso, el grupo se llama webservers.

vars: es una lista de variables expresadas como clave-valor.

remote_user: indica el usuario que ejecutará las tareas en el host remoto.

[38] http://www.yaml.org/start.html
[39] http://jinja.pocoo.org/docs/2.9/

tasks: es el listado de tareas que se ejecutarán.

handlers: las tareas pueden notificar la ejecución de alguna acción. Los *handlers* son los encargados de ejecutar estas acciones. Por ejemplo, luego de cambiar la configuración de un servicio, la tarea puede solicitar el reinicio de ese servicio; el *handler* es quien escucha esa solicitud y reinicia el servicio.

Los *playbooks* se pueden tornar demasiado extensos y seguramente se den casos de código repetido. Por esta razón Ansible provee un mecanismo para desagrupar los elementos de un *playbook* a la vez que se facilita el reuso de código: roles. Los roles permiten cargar automáticamente archivos de variables, tareas, *handlers* y otras componentes más dispuestas en una estructura de archivos bien definida.

Las componentes básicas de un rol son:

files: son archivos que las tareas copian tal cual. No se procesan, como ocurre con los *templates*. Por ejemplo, el archivo *resolv.conf* de un servidor podría ser de este tipo.

templates: aquí se encuentran los archivos de *template* o plantillas escritos en Jinja2. Las variables en **vars** y **defaults** están accesibles para los templates.

tasks: son las tareas del *playbook*, agrupadas en una sección. Las tareas también puede acceder las variables definidas en **vars** y **defaults**.

handlers: son los mismos *handlers* que estaban en el *playbook*, pero ahora agrupados en su propia sección.

vars: son las variables del *role*.

defaults: son las variables por defecto del *role*. Tienen las prioridad más baja de entre todas las variables.

meta: metadatos acerca del *role*. Por ejemplo, con metadatos se puede indicar si el *role* depende de otro o si espera correr en un sistema operativo en particular.

Todas esas secciones corresponden a un directorio en particular, de modo que la estructura de directorios para un rol llamado *webservers* sería similar a la mostrada en la Figura-8.2.

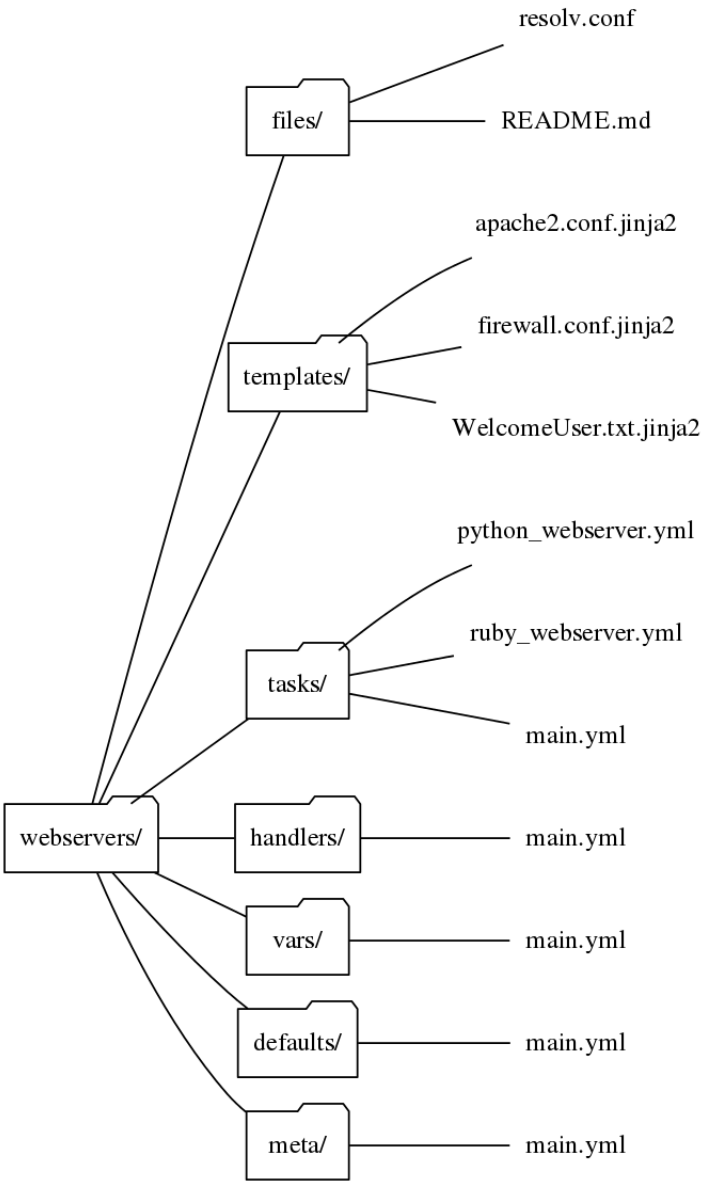

Figura 8.2: estructura de directorios de un *role* Ansible

Por último, Ansible soporta una gran variedad de herramientas relacionadas con computación en la nube[40]. En particular, dispone de módulos tanto para AWS[41] como para Proxmox[42].

8.4. Otras herramientas utilizadas

En la implementación de la arquitectura se han utilizado una gran variedad de herramientas, entre las que destacan SSH y Let's Encrypt. La primera es una herramienta que implementa el protocolo del mismo nombre y utilizada para conexiones remotas con servidores. Por otra parte, Let's Encrypt es un servicio de CA (Certification Authority o Autoridad Certificante) gratuito utilizado en Cientopolis para la implementación de HTTPS en las aplicaciones.

8.4.1. Sobre el uso de SSH

El acceso a los servidores de Cientópols se realiza exclusivamente por SSH utilizando un par de claves asimétricas[43]. En la criptografía asimétrica se utilizan dos claves, una privada y la otra pública. En el caso particular de las conexiones SSH con clave pública y privada, la clave privada reside en el sistema desde el cual se va a hacer el *log in*; mientras que la pública reside típicamente en el archivo *$HOME/.ssh/authorized_keys* del host destino.

La utilización de un *bastion host* para acceder a los servidores en la subred privada hace necesario el uso de un agente SSH. El agente de SSH *recuerda* las claves privadas del usuario, de modo que al momento de conectarse desde el *bastion host* a un *host* con IP privada, el agente utiliza la clave privada del usuario (de esta forma se evita dejar la clave privada del usuario en el *bastion host*).

Por ejemplo, si la siguiente configuración de SSH se encuentra en el archivo **ssh_deployment.conf**:

[40] http://docs.ansible.com/ansible/latest/list_of_cloud_modules.html
[41] https://www.ansible.com/aws
[42] http://docs.ansible.com/ansible/latest/proxmox_module.html
[43] Es decir, no se utiliza una contraseña de acceso, si no claves pública y privada

Listing 3: archivo de configuración para SSH

```
Host 172.16.20.235
       User ubuntu
       ProxyCommand ssh ubuntu@34.208.37.60 -W %h:%p
Host 34.208.37.60
        Hostname 34.208.37.60
        User ubuntu
        IdentityFile /home/matias/.ssh/matias-amazon
        ControlMaster auto
        ControlPath ~/.ssh/ansible-%r@%h:%p
        ControlPersist 5m
```

Entonces el usuario puede ejecutar los siguientes comandos para conectarse al host **172.16.20.235**[44] a través del host **34.208.37.60**:

```
USER_KEY="$HOME/.ssh/matias-amazon"
ssh-add -k "$USER_KEY"
ssh -F ssh_deployment.conf 172.16.20.235
```

Esencialmente, se utiliza el host con IP 34.208.27.60 como proxy para acceder al host con IP (Privada) 172.16.20.235.

La Figura-8.3 ilustra una conexión como esta, que muestra tres equipos:

Notebook: es la computadora del administrador. En ella se encuentra la clave privada, típicamente en un archivo como *$HOME/.ssh/clave_privada*.

BastionHost: es el host que actúa como proxy y tiene la correspondiente clave pública, típicamente en *$HOME/.ssh/authorized_keys*.

[44]Notar que no es necesario que la IP privada sea alcanzable directamente por el host del usuario: es suficiente con que sea alcanzable desde el bastion host.

Figura 8.3: ejemplo de conexión SSH usando un bastion host. Notar que la clave privada sólo está en la *notebook* del administrador.

PrivateHost: es el host destino que también tiene la correspondiente clave pública, generalmente en *$HOME/.ssh/authorized_keys*.

Notar que, como se dijo antes, al utilizar el agente de SSH se evita dejar la clave privada en el host que actúa de intermediario. La **conexión2** en la Figura-8.3 se realiza utilizando la clave privada que el usuario tiene en su *notebook* y no una que se encuentra en el *bastion host*.

8.4.2. Sobre el deployment de las aplicaciones

La arquitectura es agnóstica en cuanto a la manera que se hace el *deploy* de las aplicaciones. Esta decisión se fundamente en dos puntos:

- Las aplicaciones se desarrollan por diversos grupos, cada uno con su política de trabajo y de *deployment* de las aplicaciones desarrolladas. La arquitectura debe dar la posibilidad de usar diversos mecanismos de *delpoyment* pero sin obligar a los desarrolladores a utilizar uno en particular.

- Las tareas propias del *deployment* de una aplicación las conocen sus desarrolladores.

Dejando aquello en claro, se sugiere el uso de Capistrano[45] para automatizar el *deployment* de las aplicaciones. Capistrano es una herramienta de software libre escrita en ruby que automatiza diversas tareas relacionadas con el deployment de una aplicación web, como por ejemplo crear enlaces simbólicos o hacer cambios en bases de datos.

[45] http://capistranorb.com/

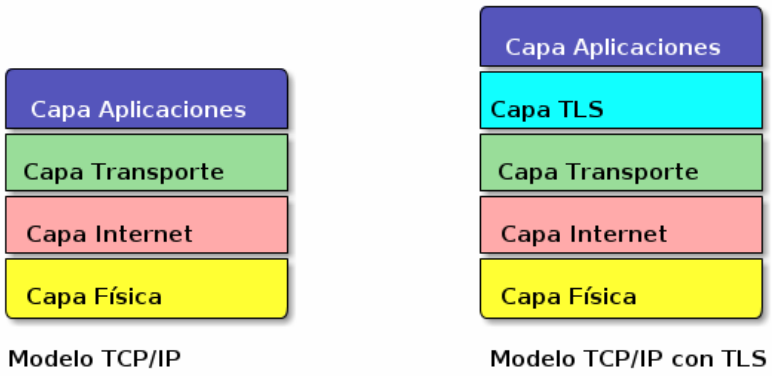

Figura 8.4: Diagrama que muestra el modelo de capas TCP/IP junto con TLS.

8.4.3. Sobre TLS y el servicio de Let's Encrypt

De acuerdo a su definición, el protocolo HTTP no soporta cifrado de datos. El soporte de encriptación para las aplicaciones web se realiza utilizando SSL/TLS como canal cifrado sobre el cual se envía el tráfico HTTP. La Figura-8.4 compara la pila TCP/IP con y sin TLS, mientras que la Figura-8.5 muestra la misma comparación pero usando el caso de HTTP exclusivamente.

La empresa Netcraft mantiene varias estadísticas relacionadas con la web, entre ellas el uso de SSL/TLS[46]. La Figura-8.6 (Tomada del sitio de Netcraft) muestra el decaimiento en el uso de SSL y el incremento de TLS a partir de fines de 2014, principalmente por problemas de seguridad en torno a SSL[47]. En junio de 2015 SSL 3.0 quedó *deprecated* u obsoleto en favor de TLS 1.2[48].

Como dice en su sitio[49], *"Let's Encrypt es un autoridad certificante gratuita, automatizada y abierta"* y está patrocinada por varias empresas del medio, como Mozilla, Akamai, Cisco y DigitalOcean. Tiene como finalidad facilitar la creación, configuración y mantenimiento de certificados TLS para masificar el uso de HTTPS.

[46] https://www.netcraft.com/internet-data-mining/ssl-survey/

[47] Ver, por ejemplo, la vulnerabilidad denominada CVE-2014-3566/POODLE en https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566

[48] https://tools.ietf.org/html/rfc7568

[49] https://letsencrypt.org/

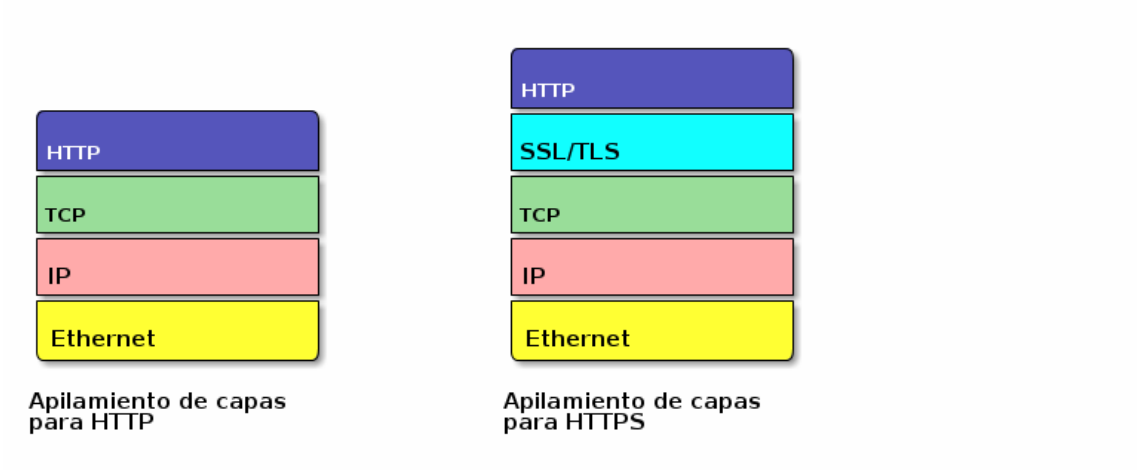

Figura 8.5: Diagrama que muestra el modelo de capas TCP/IP con TLS para HTTP.

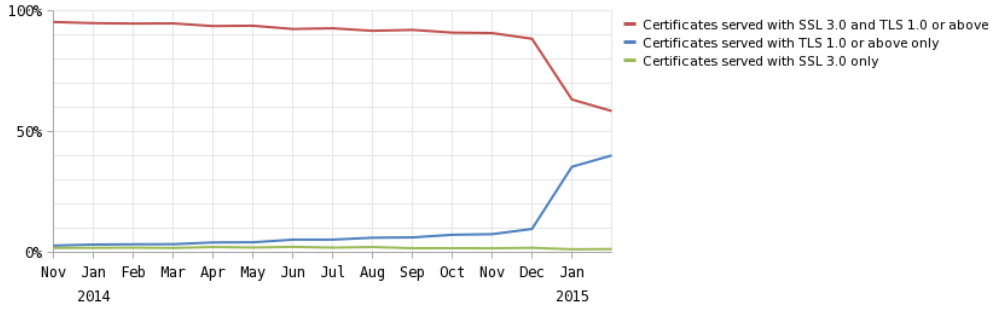

Figura 8.6: Gráfico tomado del sitio de Netcraft que muestra el avance de TLS sobre SSL

Let's Encrypt provee de certificados TLS gratuitos al proyecto Cientopolis. Cada aplicación de Cientopolis tiene su propio certificado TLS configurado en el proxy reverso. Al momento de escribir estas lineas los certificados de Let's Encrypt caducan luego de tres meses de haberse generado. Let's Encrypt utiliza el protocolo ACME[50] tanto para generar como para renovar los certificados.

8.5. Resumen

En este capítulo se mencionaron varias de las herramientas más importantes relacionadas con la computación en la nube y se describieron las tecnologías utilizadas en la implementación y gestión de la arquitectura propuesta. El siguiente capítulo muestra cómo se implementó la arquitectura de Cientópolis.

[50] https://ietf-wg-acme.github.io/acme/draft-ietf-acme-acme.html

Capítulo 9

Implementación de la arquitectura y estrategia de gestión

> Las mayores dificultades del hombre empiezan cuando puede hacer lo que quiere.
>
> *Thomas Henry Huxley*

En el capítulo 7 se mostraron las tres alternativas pensadas para actualizar y mejorar la arquitectura de Cientópolis. Utilizando LAAAM se llegó a la conclusión de que la Alternativa 2 es la que mejor se acomoda a los escenarios planteados por los *stakeholders*. En esta sección se explica como quedó implementada dicha arquitectura al momento de escribir estas lineas.

9.1. Descripción general de la implementación

En esta sección se describen las implementaciones de la arquitectura en AWS y en Proxmox. Antes de avanzar en la descripción, se hará un breve repaso sobre la alternativa elegida.

La Figura 9.1 muestra la arquitectura elegida para Cientópolis. Lo primero que se aprecia es la existencia de un balanceador que atiende las peticiones en HTTP y HTTPS. Este balanceador es la interfaz pública de la arquitectura, dado que el resto de los servidores se mantienen fuera del alcance de la Internet, pudiendo, incluso, tener direccionamiento privado.

Dependiendo de la tecnología con la que haya sido desarrollada la aplicación[1], el balanceador reenviará el requerimiento web al nodo o nodos correspondientes. Por ejemplo, si se trata de una aplicación escrita con el *framework* Django (que está implementado en Python), el balanceador redireccionará todos los requerimientos de esa aplicación a aquellos nodos web que hayan sido configurados con Python.

Como se aprecia en la figura, se tiene un servidor dedicado para las bases de datos en MySQL y PostgreSQL. Además, todos los nodos web montan por NFS (*Network File System*) un disco compartido con el código y algunos datos de las aplicaciones. Como se explica en la página del manual de mount en castellano[2], "Todos los ficheros accesibles en un sistema Unix están dispuestos en un gran árbol, la jerarquía de ficheros, con la raíz en /. Estos ficheros pueden estar distribuidos sobre varios dispositivos. La orden mount sirve para pegar el sistema de ficheros encontrado en algún dispositivo al gran árbol de ficheros". En este caso se utiliza la red para montar un *file system* que se encuentra en un dispositivo remoto.

Los desarrolladores acceden al disco compartido para el *deployment* de las aplicaciones. Notar que los usuarios no tienen posibilidad de configurar nada más que los parámetros de su aplicación. Esto garantiza la estabilidad de todo el ecosistema.

9.1.1. Implementación en AWS

Se asume en este trabajo que el lector no está familiarizado con los elementos de AWS, pero también se reconoce que no es la finalidad de este trabajo explicarlos en detalle, por lo que se los va a describir sucintamente de manera tal que el lector pueda hacerse una idea intuitiva de qué se trata cada componente:

[1] Al momento de escribir estas líneas, la arquitectura soporta aplicaciones escritas en Python, Ruby, PHP o NodeJS.

[2] `https://man.cx/mount(8)/es`

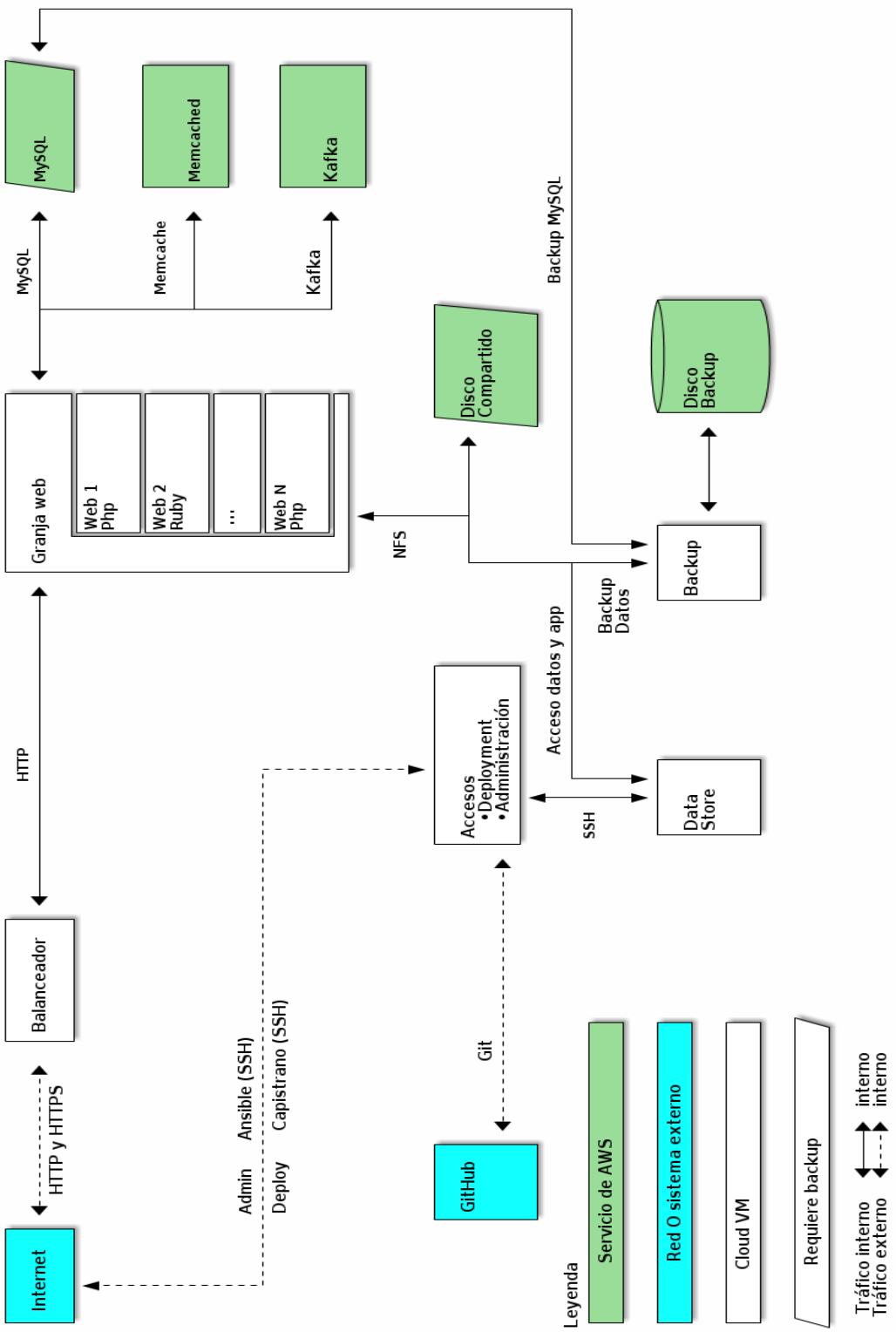

Figura 9.1: alternativa 2 para la arquitectura de Cientópolis

Region: la nube de Amazon está dividida en regiones geográficas. Tiene sentido utilizar las regiones geográficamente más cercanas al usuario final, para abaratar costos de tráfico. Además, el costo de utilizar una u otra región varía. Para el caso de este proyecto se optó por la región **us-west-2**[3] por ser de las más baratas, por estar en América y por ofrecer todos los servicios que va a consumir Cientópolis.

AZ: cada región está dividida en varias zonas de disponibilidad (*availability zones* en inglés). Se podría pensar a cada AZ como un centro de datos (*data center* en inglés) físico y a una región como el grupo de todos esos *data centers* físicos (o AZ) que la componen.

AMI: las máquinas virtuales son instancias de *templates* o plantillas ya existentes en Amazon, las AMIs[4]. Cada usuario puede generar su propia AMI, pero para el proyecto se optó por utilizar AMIs provistas por la comunidad (En particular, las máquinas virtuales son instancias de **ami-cd2147ad**[5]).

VPC: se puede decir que un VPC es un *data center* virtual, donde los administradores configuran diferentes subredes, tablas de ruteo, etc.

Security Group: actúa como un *firewall*. Varias instancias pueden ser configuradas con el mismo *security group*. En el caso de Cientópolis, por ejemplo, todos los nodos web comparten el mismo *security group*. Es decir, comparten las mismas reglas de firewall.

Subredes: el concepto de subred funciona de la misma manera que en las redes físicas. En la Figura-9.2 se muestra la red *Subred Pública 172.16.50.0/24*. No se trata de una contradicción (debido a que esa es una IP privada), sino que es la manera de designar a la red cuyos servidores tienen dos interfaces de red, una con IP pública de Amazon y otra privada, de dicha subred.

Disco de datos: el disco de datos que se menciona en la Figura-9.2 utiliza la tecnología EFS[6] de AWS.

[3] https://docs.aws.amazon.com/general/latest/gr/rande.html
[4] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html
[5] https://cloud-images.ubuntu.com/query/xenial/server/released.txt
[6] https://aws.amazon.com/efs/

NAT Gateway: es un elemento de AWS que actúa como un verdadero *NAT Gateway*, permitiendo a los servidores con IP privada alcanzar Internet, principalmente para instalar actualizaciones.

Internet Gateway: es la IP pública para todo el VPC.

Tablas de ruteo: las tablas de ruteo, que no se muestran en el dibujo, actúan como las tablas de ruteo de cualquier host y pueden ser compartidas por varias instancias. En este caso, las instancias que solamente tienen direccionamiento privado pueden acceder a Internet para descargar actualizaciones ya que tienen asignada una tabla de ruteo que indica que el *default gateway* es el *NAT Gateway*.

La Figura-9.2 muestra los elementos de infraestructura utilizados en AWS para la arquitectura de Cientópolis. Existen muchos elementos de AWS que no hacen a la arquitectura en sí, pero que de todas maneras hay que utilizarlos, como por ejemplo, VPCs, NAT Gateways, etc. La mayoría de estos elementos se pueden identificar en la Figura-9.2.

La Figura-9.3 muestra la implementación de la arquitectura utilizando AWS. En esta figura se omiten los conceptos propios de AWS como *Security groups* o *VPCs* para mejorar la legibilidad del diagrama. Con fondo blanco están aquellos elementos que son instancias de una AMI pero que se configuran utilizando Ansible. Con fondo de color verde se muestran los servicios brindados por la arquitectura pero cuya implementación está dada por AWS. También se señalan donde existen datos que deben estar resguardados o *backupeados*. Nótese, sobre esto último, que solamente se deben resguardar aquellas instancias donde están los datos de los usuarios, el disco compartido y las bases de datos, pues la configuración se puede regenerar cuantas veces sea necesario sin recurrir a los *backups*[7]. A continuación se describen los elementos de la Figura-9.3 que conforman la arquitectura implementada:

[7]Este diagrama no considera la infraestructura necesaria para programar dicha arquitectura, es decir, todos aquellos elementos usados para diseñar e implementar la arquitectura, como son, por ejemplo, el servidor de git donde se almacena el código Ansible o la wiki donde se detalla la documentación. Estos elementos deberán tener a su vez sus propias políticas de *backup*, monitoreo, etc

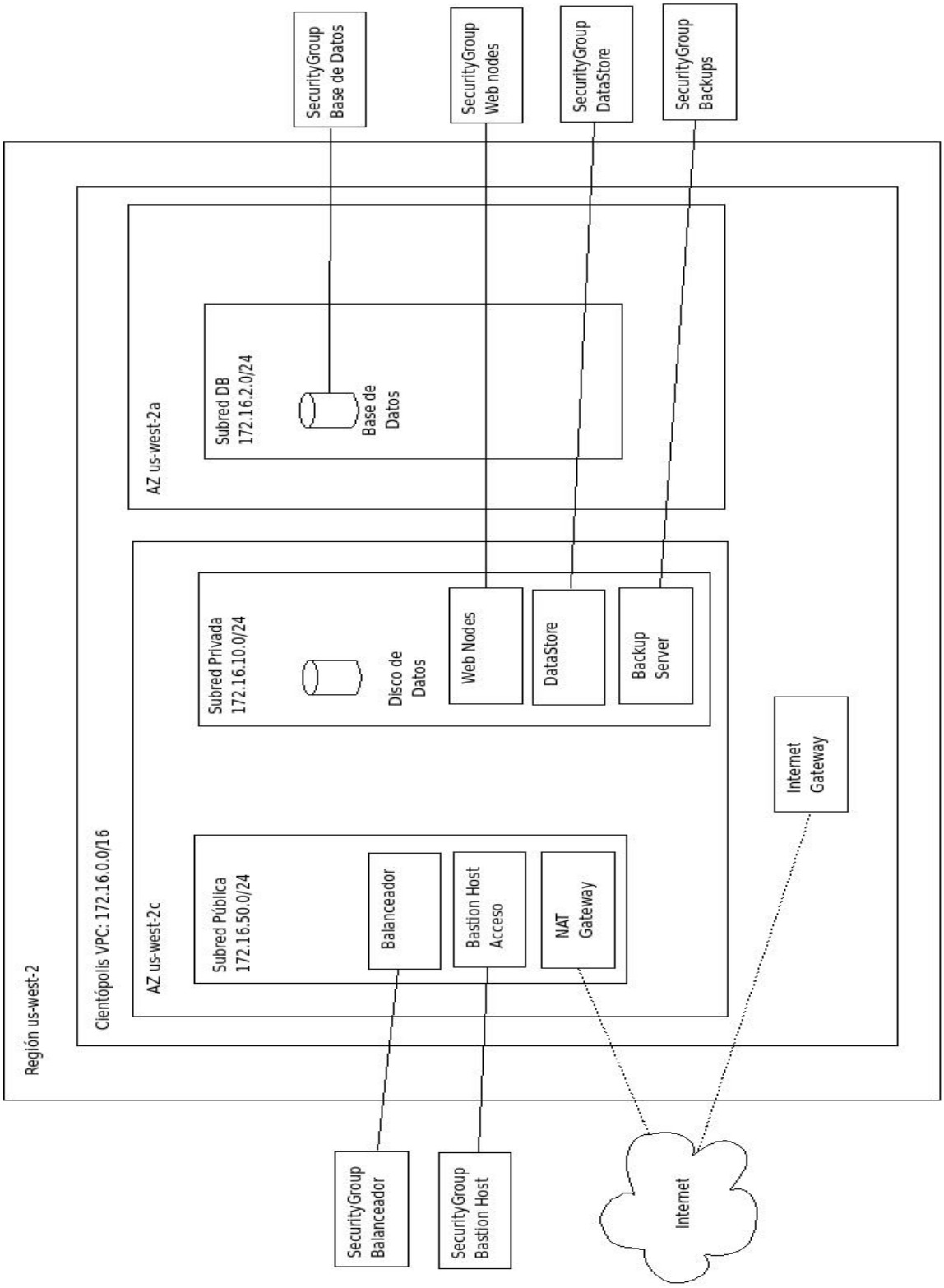

Figura 9.2: infraestructura de AWS para la arquitectura de Cientópolis

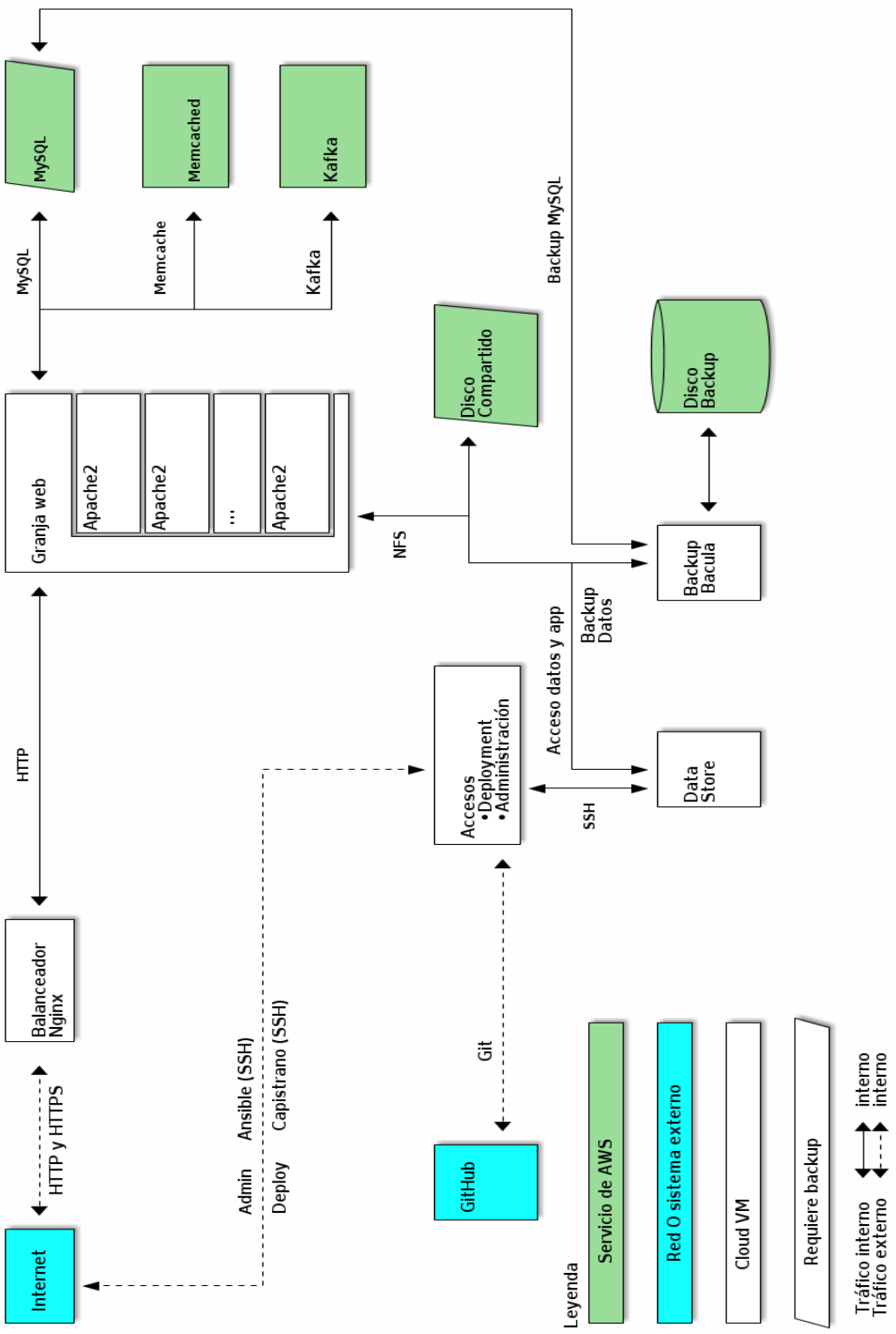

Figura 9.3: Implementación en AWS de la arquitectura de Cientópolis

Balanceador Nginx: se trata de un servidor web que actúa como proxy reverso y como balanceador de carga con una política *round robin*. También es el encargado de negociar TLS para soportar HTTPS. Por último, está configurado para redireccionar todo el tráfico HTTP a HTTPS, de forma tal que se asegura que todo el tráfico viaje cifrado. Se utiliza Let's Encrypt como autoridad certificante.

Granja web: es el conjunto de todos los servidores web que harán el trabajo de procesar las peticiones de los usuarios. Cada nodo está especializado en una sola tecnología web, como ser Ruby, Python, NodeJS, etc. Es responsabilidad del proxy reverso descrito anteriormente direccionar las peticiones del usuario al nodo web o *backend* correspondiente.

MySQL: representa todas las bases de datos de Cientópolis. En principio se utilizará MySQL, pero de forma similar se puede utilizar PostgreSQL o cualquier otra soportada por AWS.

Memcache: es una base de datos en memoria para datos de tipo clave-valor. Está a disposición de los desarrolladores para que la utilicen. Además, la misma arquitectura la usa para almacenar las sesiones de los usuarios web. Esto último es necesario debido a que el balance de carga es *round robin* y de otra manera se perdería la sesión del cliente.

Kinesis: es el servicio de *streaming* de datos propio de Amazon, donde se tienen productores y consumidores. La implementación de Proxmox utiliza Kafka.

Accesos: es un servidor que cumple dos propósitos:

1. Es adonde se conectan los desarrolladores para el *deployment* y actualización de sus aplicaciones. Para realizar estas tareas los desarrolladores pueden utilizar sistemas tipo **Capistrano** y deben acceder vía SSH autenticando con clave pública-privada.

2. Es a través de este host que los administradores pueden llegar a todos los demás servidores del ambiente, actuando como un *bastion host*. El acceso se realiza a través de Ansible, usando clave pública-privada de SSH. Si bien es posible que accedan directamente por SSH, se desaconseja en favor de utilizar Ansible solamente.

Backup: es una instancia con Bacula instalado. Se eligió este *software* debido a que el personal de Cientópolis está familiarizado con él. Además, es de fácil automatización y la curva de aprendizaje es leve.

Disco compartido: utilizando la tecnología EFS de AWS se crea un disco *muy grande* que actúa como repositorio compartido de las aplicaciones y datos. Ese disco se monta por NFS en los nodos web, de modo que cada nodo dispone del código de las aplicaciones y sus datos. También por NFS se monta en la instancia accesos, adonde se conectan los desarrolladores para subir y actualizar las aplicaciones. Por último, también se monta en el servidor de *backup*, para facilitar el proceso de resguardo de los datos.

Disco de Backup: es el medio donde se almacenan los backups. Para esto se utiliza S3 de AWS[8], por abaratar los costos y porque está diseñada para este uso en particular.

Creación de la arquitectura en la nube

El proceso de creación consta de los siguientes pasos:

1) Instanciación de la infraestructura: en primer lugar se debe instanciar la infraestructura necesaria, como *security groups*, *VPCs* o subredes.

2) Creación de las instancias: esto involucra la creación de todas las instancias requeridas: proxy reverso, nodos web, servidor de backup, etc.

[8] https://aws.amazon.com/s3/

3) Configuración inicial: luego se debe configurar el acceso por SSH desde la máquina local, aquella ejecutando Ansible y que es desde donde trabajará el administrador, hacia las instancias recién creadas. Esto se realiza creando una configuración particular para SSH. También en este paso se configura Python2 en las instancias[9].

4) Configuración final: por último se configura cada instancia según corresponda, se crean los usuarios correspondientes a cada aplicación y se crean las bases de datos.

Sobre una arquitectura existente, el único paso que deberá repetir el administrador es el último. Por ejemplo, si debe dar de alta una nueva aplicación, deberá agregarla a la configuración de Cientópolis y ejecutar ese último paso.

9.1.2. Implementación en Proxmox

La implementación en Proxmox está sujeta a ciertas restricciones relacionadas más con el ambiente en el que ejecuta Proxmox que con Proxmox en sí mismo. Por ejemplo, la creación de máquinas virtuales se hace manualmente y bajo demanda, contrario a como se realiza en Amazon o cualquier otro servicio en la nube. Esto se debe no a una limitación de Proxmox, sino a al manera en que se utiliza en el ambiente de trabajo de Cientópolis.

La implementación en Proxmox debe implementar algunas funciones que en Amazon vienen dados como servicios. Por ejemplo, mientras que en Amazon ofrece como servicios la base de datos o de DNS, cuando se trabaja con el ambiente de Proxmox se deben implementar esos servicios. En particular, la arquitectura en Proxmox configura:

- Servicio de MySQL
- Servicio de PostgreSQL
- Servicio de DNS

[9] Ansible trabaja mejor con Python2 y la AMI usada tiene instalado Python3, por lo que se utiliza Ansible para instalar Python2 en cada instancia.

Y, por otra parte, hay conceptos que en Amazon deben manejarse y que en Proxmox directamente no existen, como son:

- VPC
- Security Groups
- Ruteo, NAT e IPs públicas

Como el lector habrá notado, existen dos capas en la implementación de la arquitectura. La primera viene dada por la instanciación de todos los servicios necesarios para que las aplicaciones funciones, como Apache, MySQL, etc. La segunda se encarga de crear el espacio, los usuarios y las bases de datos necesarias para las aplicaciones. Por ejemplo, la primera capa instala el servicio de MySQL, mientras que la segunda crea la base de datos y configura los permisos apropiados para cada aplicación.

Por otra parte, el ambiente ya cuenta con soluciones de backup y monitoreo, por lo que no son necesarias implementarlas[10].

Por último, el acceso al servicio de Proxmox es restringido y el autor de este trabajo no cuenta con acceso directo, sino que debe solicitar la creación de las máquinas virtuales al operador del servicio.

Creación de la arquitectura en Proxmox

En la sección anterior se mencionaron los pasos para crear la arquitectura propuesta sobre Amazon. En esta sección se comenta el trabajo necesario para instanciar la arquitectura sobre Proxmox, siguiendo los pasos usados en Amazon como guía:

Instanciación de la infraestructura: en Amazon se deben instanciar elementos como *security groups*, *VPCs* o subredes. Sin embargo, en el ambiente de Proxmox algunos de estos conceptos no existen, como es el caso de los *security groups* o el administrador ya los instanció para este u otro proyecto, como es el caso de las subredes. Por esta razón, el código para crear el ambiente en Proxmox no implementa la instanciación de la infraestructura.

[10]Cabe aclarar que en este caso, se deberá instalar en los servidores los clientes o agentes requeridos por las soluciones en particular.

Creación de las instancias: en el caso de Amazon, el segundo paso consiste en crear todas las instancias requeridas: proxy reverso, nodos web, servidor de backup, etc. Para el caso de Proxmox, como se dijo antes, es el administrador de Proxmox quien crea las instancias. Esto no se debe a una limitación de Proxmox (bien se podrían crear las máquinas virtuales programáticamente), sino a la manera de utilizar la herramienta en el contexto donde se realizó el trabajo.

Configuración inicial: luego se debe configurar el acceso por SSH desde la máquina local, aquella ejecutando Ansible y que es desde donde trabajará el administrador, hacia las instancias recién creadas. Esto se realiza creando una configuración particular para SSH. También en este paso se configura Python2 en las instancias[11].

Configuración final: por último se configura cada instancia según corresponda, se crean los usuarios correspondientes a cada aplicación y se crean las bases de datos.

En el caso de la arquitectura instanciada en Proxmox, el acceso a los servidores se hace mediante una VPN, por lo que no es necesario usar una *bastion host*.

Sobre una arquitectura existente, el único paso que deberá repetir el administrador es el último. Por ejemplo, si debe dar de alta una nueva aplicación, deberá agregarla a la configuración de Cientópolis y ejecutar ese último paso.

9.2. Descripción del código Ansible

En esta sección se explica el código escrito para la implementación de la arquitectura de Cientopolis. El código completo se encuentra en el DVD adjunto a este trabajo y en GitHub[12].

[11] Como se dijo antes, Ansible trabaja mejor con Python2 y las máquinas virtuales creadas en Proxmox tienen instalado Python3, por lo que se utiliza Ansible para instalar Python2 en cada instancia. Para más información acerca del soporte de Ansible para Python3 puede visitar https://docs.ansible.com/ansible/latest/reference_appendices/python_3_support.html

[12] https://github.com/cientopolis/architecture.git

9.2.1. Sobre los directorios y archivos

En el directorio raíz del proyecto se encuentran los siguientes subdirectorios:

bin: debido a la manera en que el código está parametrizado, se debe ejecutar Ansible con varios parámetros. En este directorio se encuentra un *script* que funciona como *wrapper*, para que la ejecución de Ansible sea más amigable.

config: en este directorio se encuentran archivos de configuración para Ansible y para SSH.

deployments: en principio se tienen dos ambientes o entornos, uno de prueba (*testing*) y otro de producción. Los datos de configuración de cada uno de esos entornos se encuentran en distintos archivos dentro de este directorio. Podrían existir más entornos, no hay una limitación para tener sólo estos dos.

doc: aquí se encuentra la documentación de la arquitectura y tutoriales para el uso de las herramientas.

include: en este directorio reside la lógica de la aplicación. Este directorio contiene algunos subdirectorios:

files: archivos que se copian con Ansible a los servidores de Cientópolis.

roles: roles de Ansible. Al momento de escribir estas lineas se usan dos roles de terceros: **ansible-role-certbot**[13] y **rvm1-ansible**[14]; el primero creado por Jeff Geerling y el segundo por Nick Janetakis. Si existiera la necesidad de escribir un role de Ansible para Cientópolis, debería residir en este directorio. En el archivo de configuración de Ansible se indica que este es el directorio adonde debe buscar los roles.

tasks: algunas tareas que son comunes a la instanciación en AWS y Proxmox se modularizaron en este directorio. Por ejemplo, una vez creada la instancia en AWS o la máquina virtual en Proxmox, la configuración de un nodo PHP es la misma en ambos entornos. Esas tareas que son comunes en ambos entornos están modularizadas en este directorio.

[13] https://github.com/geerlingguy/ansible-role-certbot
[14] https://github.com/rvm/rvm1-ansible

templates: aquí se guardan los templates en jinja2[15] de usados en las tareas de Ansible.

ec2.ini y ec2.py Cuando se utiliza AWS, el inventario de Ansible es dinámico. Con estos dos archivos se logra la consulta de las instancias en Amazon y la generación del inventario dinámico para usarlo en Ansible. Estos archivos se usan sólo cuando la arquitectura se instancia en AWS.

README.md Brinda una descripción básica de los archivos en el proyecto y de cómo utilizarlos.

vault_password.txt Los datos con información sensible se encuentran cifrados usando una tecnología de Ansible llamada *vault*. En este archivo, que no se versiona, se guarda la contraseña para descifrar esos datos. Opcionalmente, si no se desea guardar este archivo, se puede configurar Ansible para que solicite la contraseña en cada ejecución.

9.2.2. Ejemplo: configuración del DNS

En esta sección se muestra cómo se configuran los servidores de DNS cuando Cientópolis se instancia en Proxmox. Se asume que las dos máquinas virtuales para los servidores de DNS primario y secundario están creadas y se tiene acceso por SSH.

El *wrapper* para ejecutar Ansible en la infraestructura de Proxmox es **bin/init_cientopolis_deployment_proxmox.sh**. A continuación el código 4 muestra un extracto de dicho script:

Listing 4: configuración de DNS

```
1 #!/bin/bash
2 # [prod|testing]
3 DEPLOYMENT_NAME="$1"
4 USER_NAME="$2"
5
```

[15] http://jinja.pocoo.org/

```
6  ANSIBLE_CONFIG="config/ansible_$DEPLOYMENT_NAME.cfg"
7  SSH_CONFIG="config/ssh_$DEPLOYMENT_NAME.config"
8
9  INVENTORY="deployments/$DEPLOYMENT_NAME/inventory"
10
11 VARS="-e @deployments/$DEPLOYMENT_NAME/apps.yml -e \
12         @deployments/$DEPLOYMENT_NAME/dbs.yml -e  \
13         @deployments/$DEPLOYMENT_NAME/vars.yml -e \
14         @deployments/$DEPLOYMENT_NAME/vault.yml"
15
16 echo "Init servers :: Inicializando las instancias"
17 ANSIBLE_CONFIG="$ANSIBLE_CONFIG" ansible-playbook $VARS \
18                -i $INVENTORY -u $USER_NAME -b \
19                --vault-password-file=vault_password.txt \
20                -e 'ansible_python_interpreter=/usr/bin/python3'
21                "include/init_proxmox.yml"
22
23 echo "Init servers :: Configuracion Basica"
24 ANSIBLE_CONFIG="$ANSIBLE_CONFIG" ansible-playbook $VARS \
25                -i $INVENTORY -u $USER_NAME -b \
26                --vault-password-file=vault_password.txt \
27                "include/config_basic_proxmox.yml"
28
29 echo "Configurando DNS server"
30 ANSIBLE_CONFIG="$ANSIBLE_CONFIG" ansible-playbook $VARS \
31                -i $INVENTORY -l dns -u $USER_NAME -b \
32                --vault-password-file=vault_password.txt \
33                "include/config_dns_proxmox.yml"
34 ...........
```

El archivo **bin/init_cientopolis_deployment_proxmox.sh** es un script ejecutable en Bash que tiene por función correr Ansible con más facilidad para el administrador.

En la linea 2 del código mostrado aparecen comentados los dos ambientes de ejecución: *prod* y *testing*. Como se explicó anteriormente, cada uno de estos ambientes tiene su propio directorio. En las lineas 3 y 4 se nombran los dos argumentos con los que se invoca el script. Por ejemplo, para el script sobre el ambiente de *producción* con el usuario *matias*, la invocación sería así:

bin/init_cientopolis_deployment_proxmox.sh prod matias

Notar que tanto el inventario como los archivos de variables dependen de la variable **DEPLOYMENT_NAME**. El primer llamado a Ansible se realiza en la linea 17 y tiene por objetivo instalar las dependencias básicas en las máquinas virtuales. En este caso se asegura que python2 esté instalado.

La segunda ejecución de Ansible aparece en la linea 24, donde se realizan varias tareas esenciales, como instalar paquetes (vim, rsyslog, ntp y varios más) y agregar los usuarios del sistema.

En la linea 30 se ejecuta Ansible para configurar los servidores de DNS. Notar el modificador *-l dns*, indicando que de entre todos los servidores en el inventario, se ejecute sólo en aquellos que se encuentran en el grupo llamado *dns*. En este caso el archivo que se lee es **include/config_dns_proxmox.yml**, que es el playbook donde se configuran los servidores de DNS.

El archivo **include/config_dns_proxmox.yml** se muestra en el código 5:

Listing 5: playbook de Ansible que configura el DNS

```
1  ---
2  - hosts: "all"
3    gather_facts: False
4    tasks:
5      - name: Install Bind packages
6        apt:
7          name: '{{ item}}'
8          state: 'present'
9        with_items:
10         - 'bind9'
11         - 'bind9utils'
```

```
12          - 'bind9-doc'
13      - name: Create serial, based on last two digits
14              of year, month, day, hour and minute
15        command: date +%y%m%d%H%M
16        register: dns_serial
17        changed_when: false
18        run_once: true
19      - name: Updating /etc/default/bind9
20        template:
21          src: default.bind9.jinja2
22          dest: /etc/default/bind9
23          owner: root
24          group: root
25          mode: 0644
26      - name: Updating /etc/bind/named.conf.local
27        template:
28          src: named.conf.local.jinja2
29          dest: /etc/bind/named.conf.local
30          owner: root
31          group: root
32          mode: 0644
33      - name: Updating zone file
34        template:
35          src: zone.jinja2
36          dest: "/etc/bind/db.{{ cientopolis_main_zone }}"
37          owner: root
38          group: root
39          mode: 0644
40        notify:
41          - Restart DNS
42    handlers:
43      - name: Restart DNS
```

```
44      service: name=bind9 state=restarted
```

Lo primero que se debe advertir es el idioma de los scripts. Se buscó escribir los nombres de variables y comentarios en el idioma inglés para que se más fácil compartir el código con personas de otros países.

En la linea 2 se indica que se va a trabajar sobre todos ("all") los hosts. Pero como en la invocación (linea 30 del script anterior) se filtra por el grupo *dns* (*-l dns*), se termina ejecutando en sólo los hosts del grupo dns.

La primer tarea (linea 5) es instalar los paquetes de Bind, que es el servidor de DNS elegido para servir las zonas de Cientópolis. En la linea 13 se crea el número de serie para la zona[16].

En las lineas 18 y 25 se actualiza la configuración de Bind, mientras que en la linea 32 se actualiza la zona de Cientópolis. El template para la zona es el archivo **zone.jinja2**, que se muestra en el código 6.

Listing 6: playbook de Ansible que configura el DNS

```
1  $TTL        300
2
3  @        IN        SOA        {{ cientopolis_ns1 }}. \
4                              root.{{ cientopolis_main_zone }}. \
5                              ( {{ dns_serial.stdout }} 1800 900 604800 86400 )
6          IN        NS        {{ cientopolis_ns1}}.
7          IN        NS        {{ cientopolis_ns2}}.
8
9  ns1.{{ cientopolis_main_zone }}.     IN      CNAME    {{ cientopolis_ns1 }}.
10 ns2.{{ cientopolis_main_zone }}.     IN      CNAME    {{ cientopolis_ns2 }}.
11
12 {% for anApp in applications %}
13   {% if anApp.enabled == True %}
14     {% if anApp.domain != cientopolis_loadbalancer %}
```

[16]Toda zona de DNS tiene un número de serie; cuando se actualiza una zona, su número de serie debe ser mayor que la que tenía antes. Usando el formato de año, mes, día, hora y minuto se asegura esto, asumiendo que existe un cambio por minuto como máximo.

```
15        {{ anApp.domain }}.       IN      CNAME    {{ cientopolis_loadbalancer }}.
16      {% endif %}
17    {% endif %}
18  {% endfor %}
19
20  {% for anApp in applications %}
21    {% for anAlias in anApp.aliases %}
22      {% if anApp.enabled == True %}
23        {% if anApp.domain != cientopolis_loadbalancer %}
24          {% if anAlias != cientopolis_loadbalancer %}
25            {{ anAlias }}.        IN      CNAME    {{ cientopolis_loadbalancer }}.
26          {% endif %}
27        {% endif %}
28      {% endif %}
29    {% endfor %}
30  {% endfor %}
```

Debido a que el archivo es un *template*, hay una mezcla de sintaxis. Por un lado se tiene la sintaxis propia del archivo de configuración de Bind y por otro se tienen secciones escritas en Jinja2, que es el lenguaje usado por Ansible para escribir los templates. Las lineas escritas en Jinja2 se procesan y dan como resultado lineas válidas según el formato de configuración de Bind.

En las primeras lineas del archivo se configura la zona de Cientópolis. Por ejemplo, en las lineas 6 y 7 se indican cuáles son los servidores de nombre para la zona.

En la linea 12 empieza un bucle en donde se itera sobre las aplicaciones definidas y si la aplicación está habilitada, se crea una entrada en el DNS para ella. En la linea 20 empieza otro bucle que tiene por objeto iterar sobre los aliases de cada una de las aplicaciones definidas.

El procesamiento del template da como resultado un archivo de zonas válido para Bind. Cuando se reinicie Bind, leerá el nuevo archivo de zonas y esos nombres de dominio empezarán a estar disponibles.

9.3. Resumen

Se mostró la implementación de la arquitectura utilizando Ansible en Amazon y en Proxmox. Se explicaron las particularidades que se deben considerar para los dos ambientes, como por ejemplo, el hecho de que Amazon brinda MySQL como un servicio, mientras que Proxmox no, y por lo tanto, se debe crear una máquina virtual e instalar en ella el servicio de MySQL.

En el capítulo siguiente se presenta una evaluación de la arquitectura propuesta.

Capítulo 10

Evaluación de la arquitectura propuesta

> Lo que se necesita no es la voluntad de creer, sino el deseo de averiguar, que es exactamente lo contrario
>
> *Bertrand Russell*

En este capítulo se evalúa la arquitectura y los procesos de gestión propuestos y se analiza el impacto en relación a la situación original. También se comenta la experiencia que se tuvo al utilizar la metodología LAAAM para diseñar la nueva arquitectura de Cientópolis y se incluyen algunos comentarios sobre la experiencia de usuario con las herramientas utilizadas y la arquitectura implementada.

10.1. Sobre la arquitectura resultante

Una cuestión a considerar es la complejidad de la arquitectura en sí y las herramientas utilizadas en su implementación, como Nginx o MySQL. El uso de virtualización y de herramientas de aprovisionamiento como Ansible facilitan la administración del entorno, de modo tal que los desarrolladores no tienen necesidad de realizar tareas administrativas en los servidores ni de entender el diseño e implementación de la arquitectura. Los administradores, por otra parte, disponen de herramientas para la fácil administración de la arquitectura y la curva de aprendizaje se vuelve menos pronunciada para los nuevos miembros del proyecto.

En cuanto al software provisto, la nueva arquitectura mantiene mucho del software usado originalmente. Por ejemplo, como servidor web se sigue utilizando Apache, y MySQL como motor de base de datos relacional. Esto se debe a que son requisitos de las aplicaciones de Cientópolis. En este punto, la diferencia con el ambiente original es la versión de estas herramientas y algunos detalles de configuración de cada una.

Como se explica en el párrafo anterior, las herramientas de automatización empleadas mejoran la usabilidad de la arquitectura. Los desarrolladores no deben acceder más a los servidores para configurar los servicios que necesiten, y los administradores disponen de la configuración de la arquitectura en un formato versionado y en texto plano, que se auto-documenta, debido a que los *playbooks* de Ansible sirven como documentación.

La gestión de la arquitectura se vuelve más ágil, a tono con las metodologías de desarrollo usadas por los programadores de las aplicaciones de Cientopolis. También la complejidad de la administración se reduce, lo cual ayuda a mejorar el desempeño de los administradores.

La disponibilidad de un ambiente de *testing* es esencial para el mantenimiento y mejora de los *playbooks* de Ansible, de modo que no se ejecuten pruebas en el ambiente de producción. Esto implica un mayor consumo de recursos, aunque sólo durante el proceso de *testing*.

El diseño de la arquitectura es escalable horizontal y verticalmente, aunque no implementa alta disponibilidad. La separación de servicios en diferentes nodos o servidores permite escalar cada servicio individualmente según las necesidades. Hay que aclarar que este escalamiento no se produce automáticamente como ocurre en algunos proveedores de nube, como AWS, sino que lo debe implementar el administrador.

Por último, todas las aplicaciones que ejecutan en el ambiente de Cientópolis usan TLS, lo cual mejora la seguridad.

10.2. Sobre la metodología usada

La evaluación de la arquitectura se realizó ejecutando el método LAAAM y quedó demostrado objetivamente que la arquitectura elegida es la que mejor se adapta a los escenarios planteados por los *stakeholders*.

Como se mencionó en las primeras páginas de este trabajo, LAAAM es, de entre todas las metodologías de evaluación de arquitecturas, la más sencilla y *lightweight* y se lleva bien con las metodologías ágiles de desarrollo de software. Pero sobre esto se debe hacer una aclaración: la base de las metodologías ágiles es *abrazar el cambio*, por lo que la arquitectura también debe ser capaz de adaptarse rápidamente al cambio. LAAAM es ágil en lo relacionado a la ejecución del método, pero no genera necesariamente *arquitecturas ágiles*. El diseño e implementación de *arquitecturas ágiles* es trabajo del arquitecto y el método no las garantiza.

Las fortalezas del método son:

- Una forma sistemática de evaluar características y *ranquearlas*. Esto facilita la trazabilidad de los requerimientos y de sus cambios en el tiempo. También funciona como una linea de defensa ante los futuros cambios en la arquitectura y los escenarios *ranqueados* sirven como *fitness functions* que protegen la arquitectura. "Las *fitness functions* de una arquitectura proveen una manera objetiva de evaluar la integridad de alguna característica de esa arquitectura" (Ford et. al, 2017).

- Es el más sencillo de ejecutar y el que menos entregables genera. Es decir, es el que menos trabajo requiere para su ejecución y mantenimiento.

También el método tiene algunas debilidades:

- Depende de que los *stakeholders* se reúnan y colaboren con el desarrollo de la arquitectura. De todas maneras, esa es una necesidad para todos los aspectos del proyecto y no sólo para el éxito de la arquitectura.

- Dado que la arquitectura no es estática, sino que es una cosa cambiante, es necesario mantener los escenarios y la documentación. No hay manera de automatizarlo. Esto agrega carga de mantenimiento y quita recursos para otras tareas. Para proyectos "pequeños", quizás no tenga sentido utilizar esta metodología. En el capítulo siguiente se vuelve sobre este tema.

10.3. Sobre las herramientas usadas

Como herramienta de aprovisionamiento de servidores se utilizó Ansible, que resultó ser de fácil uso y mantenimiento. Debido a que no requiere instalar ningún agente en el servidor, su uso no quita recursos a las aplicaciones ni expone potenciales fallos de seguridad o vulnerabilidades.

Quizás lo más complicado de Ansible es entender que se trata de una herramienta con la cual se dice o describe lo que se debe hacer, pero no como hacerlo, similar a otras herramientas declarativas, como el lenguaje SQL.

Por otra parte, dado que se usa YAML, se debe tener precaución a la hora de usar espacios o TABs. Sin embargo, este aspecto de la sintaxis de YAML no ha causado demasiados inconvenientes, y los pocos errores de sintaxis fueron fácilmente resueltos.

Sobre las infraestructuras donde corre esta arquitectura, AWS es una herramienta sorpresivamente sencilla de usar si se tiene en cuenta la complejidad de las herramientas y funcionalidades que provee. Quizás como punto en contra se pueda mencionar el sistema de pago, que es muy complicado de entender si se compara con otras soluciones de *cloud computing* en donde se entiende con facilidad cuanto se va a pagar por el servicio, como es el caso de Digital Ocean.

10.4. Resumen

En este capítulo se mostró una evaluación general de las herramientas utilizadas, tratando de mencionar tanto sus aspectos positivos como negativos. Se habló sobre el método elegido (LAAAM) y sobre los aspectos técnicos de la arquitectura.

En el siguiente capítulo se mencionan las conclusiones generales, las deudas técnicas y se dan los trabajos a futuro.

Capítulo 11

Conclusiones

> Creo que el conocimiento científico posee propiedades fractales, que no importa cuánto podamos aprender; que todo lo que quede, por pequeño que pueda parecer, es tan infinitamente complejo como el total del cual partimos. Ése, creo, es el secreto del universo.
>
> *Isaac Asimov*

El objetivo general de esta tesis es evaluar, en un caso de estudio concreto, la aplicabilidad e impacto de los principios de "infraestructuras como código" en la calidad de las arquitecturas de proyectos de investigación e innovación tecnológica, y en los procesos de gestión de las mismas. La aplicación de estos conceptos se traduce en una mejora en la administrabilidad de la arquitectura, exhibiendo una mejora en los siguientes puntos:

- Menos tiempo y esfuerzo de administración de los servicios.
- Menos riesgo de romper el ambiente luego de un *deployment*.
- Ningún administrador accede directamente al ambiente de producción.
- Ningún desarrollador accede a las configuraciones de los servicios.

- Mejora en el tiempo requerido para armar un ambiente de testing o pruebas para los desarrolladores.

Se implementó exitosamente la nueva arquitectura de Cientópolis usando los conceptos de infraestructura como código y siguiendo la metodología LAAAM para elegir la mejor de las opciones arquitectónicas.

Si bien la administración de la arquitectura no es totalmente autónoma, se deben crear las máquinas virtuales en Proxmox, por ejemplo, sin dudas lo desarrollado sirve como base para llevar la gestión de la arquitectura a un modelo 100 % automatizado, empoderando tecnológicamente a las comunidades de ciencia ciudadana y achicando la brecha hacia la implementación de una politica de "soberanía tecnológica".

En cuanto a la investigación realizada, se indagó sobre el estado de arte de las metodologías usadas en la evaluación de arquitecturas, como son LAAAM, ATAM y otras, concluyendo que, para el caso de Cientópolis, era más conveniente utilizar LAAAM debido a su bajo costo de ejecución.

Se relevaron las carencias o falencias técnicas que tenía la arquitectura original y su manera de administrarla. Luego se acomodó dicha arquitectura y se la modularizó, reduciendo así la dificultad, tiempo y esfuerzo de administrarla, usando herramientas de *infraestructura como código*, Ansible en este caso.

De esta forma se considera que se cumplieron los objetivos específicos y el objetivo general de este trabajo enumerados al inicio del mismo.

11.1. Deuda técnica

El término *bit rot* se refiere a la degradación que sufren los sistemas a medida que pasa el tiempo. Habitualmente se da porque no existen guías a la hora de hacer los cambios. Este trabajo no ofrece ninguna guía ni pauta para evitar este problema[1], por lo que la arquitectura propuesta está expuesta a (y seguramente con el tiempo sufra) esta degradación. Se considera esto una deuda técnica y se plantea como trabajo a futuro la investigación e implementación de técnicas anti-degradación, como las propuestas en **Building Evolutionary Architectures** (Ford, N., Parsons, R. & Kua, P., 2017).

Los *playbooks* sirven para crear y modificar la configuración de los servidores, pero no para eliminar elementos de esa configuración. El borrado se debe hacer a mano. Sí se implementa la desactivación de aplicaciones. Por ejemplo, se puede crear, actualizar o desactivar el espacio de una aplicación, pero no borrarla. Para eliminarla, se debe hacer el borrado a mano. En otras palabras, se realiza un borrado lógico, pero no se implementó un borrado físico.

Al momento de escribir estas lineas queda pendiente la instalación y configuración de un sistema de monitoreo y alerta para la arquitectura.

11.2. Mejoras o trabajos a futuro

En esta última sección se mencionan algunos posibles trabajos a futuro:

Tamaño de proyectos: investigar las dimensiones que debe tener un proyecto para justificar el uso de LAAAM. En otras palabras, qué tan "grande" debe ser un proyecto de software para que se justifique el uso de LAAAM o alguna de las otras metodologías aquí revisadas.

[1] Como se dijo anteriormente, los escenarios trabajan como *fitness functions*, pero son sólo la punta del iceberg.

Ambientes multinube: extender la implementación resultado del presente trabajo, desarrollando una solución *multinube* o de nube híbrida. En este caso la idea sería brindar servicio desde la solución de virtualización propia complementada con una solución basada en alguna nube. Actualmente el ambiente puede ser instanciado totalmente en máquinas virtuales y parcialmente en la nube de Amazon, pero no en una mezcla de ambos.

Soporte para MQTT o similares: MQTT es un protocolo de mensajería usado ampliamente en soluciones de IoT. Actualmente Cientópolis no dispone de soporte para este protocolo, pero puede investigarse al respecto y desarrollar una.

Extensión de la plataforma: se puede extender la solución para que soporte otros sistemas operativos y arquitecturas de hardware, como Windows o FreeBSD.

Instrumentación de aplicaciones: se puede extender la arquitectura para brindar servicios de monitoreo, utilizando Prometheus[2] o Riemann[3], por ejemplo.

Editor de configuraciones: al momento de escribir estas lineas la configuración de las aplicaciones y bases de datos se realiza por medio de archivos YAML editados por el administrador. Sería deseable la existencia de un editor de configuraciones para los recursos de Cientópolis. Desarrollar un editor que incorpore la semántica necesaria para configurar los servicios de Cientópolis puede ser otro trabajo a futuro.

Licenciamiento: dado que las aplicaciones y arquitectura de este proyecto de ciencia ciudadana tiene un trasfondo social, se propone como trabajo a futuro la investigación del licenciamiento de todas estas componentes (playbooks, aplicaciones, documentos varios) para su posible *release* a la Comunidad.

[2] https://prometheus.io/
[3] http://riemann.io/

Referencias

Alexander, C. (1977), *A Pattern Language*, Oxford University Press.

Barron, F. & Barret, B. (1996). Decision quality using ranked attribute weights. *Management Science, 42*(11), 1515-1523.

Bass, L., Clements, P. & Kazman, R. (2012), *Software Architecture in Practice*, Addison-Wesley.

Bass, L. & Nord, R. (2012), *Understanding the Context of Architecture Evaluation Methods*, 2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture. Congreso llevado a cabo en Helsinki, Finlandia.

Bijlsma, L. (2007), *Software architecture*, http://www.cs.uu.nl/wiki/Master/SoftwareArchitecture Accedido en octubre de 2007.
Actualmente disponible en http://web.archive.org/web/20070607172151/www.cs.uu.nl/wiki/Master/SoftwareArchitecture

Breivold, H. (2009), *Software Architecture Evolution And Software Evolvability*, School of Innovation, Design and Engineering, Mälardalen University, Suecia

Buschmann, F., Meunier, R., Rohnert, H., Sornmerlad P. & Stal, M. (1996), *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, John'Wiley & Sons Ltd.

Carriere, J. (2009), *Architecture enabling business - It's pronounced like "lamb", not like "lame"* . Recuperado el 19 de mayo de 2018 de https://technogility.wordpress.com/2009/05/11/its-pronounced-like-lamb-not-like-lame/

Caum, C. (2013), *Continuous Delivery Vs. Continuous Deployment: What's the Diff?*, Puppet blog, Recuperado el 21 de mayo de 2018 de
https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010), *Documenting Software Architectures: Views and Beyond (2nd Edition)*, Addison-Wesley.

Clements, P. & Bass, L. (2010). The Business Goals Viewpoint. *IEEE Software, 27*(6), 38-45.

Clements, P. & Bass, L. (2010a), *Using business goals to inform software architecture*, Proceedings of the 2010 18th IEEE International Requirements Engineering Conference. Congreso llevado a cabo en Washington, Estados Unidos.

Clements, P. & Kazman, R. (2001) *Evaluating Software Architectures: Methods and Case Studies* , Addison-Wesley.

Chen, L. (2015), *Towards Architecting for Continuous Delivery*, 2015 12th Working IEEE/IFIP Conference on Software Architecture. Congreso llevado a cabo en Montreal, Canadá.

Cois, C., Yankel, J. & Connell, A. (2015), *Modern DevOps: Optimizing software development through effective system interactions*. IEEE International Professional Communication Conference 2015. Congreso llevado a cabo en Limerick, Irlanda.

de Vries, T. (2008), *Architectural Knowledge in Quantitative Architectural Analysis*, University of Groningen, Holanda.

Duffy, M. (2015), *DevOps Automation Cookbook*, Packt Publishing.

Ebert, C., Gallardo, G., Hernantes, J. & Serrano, N. (2016). DevOps. *IEEE Software, 33*(3), 94-100.

Erder, M. & Pureur, P. (2015), *Continuous Architecture*, Morgan Kaufmann

Fairbanks, G. (2014). (2014, marzo 24). Architecture Patterns vs. Architectural Styles [Archivo de video]. Recuperado de
http://www.georgefairbanks.com/blog/architecture-patterns-vs-architectural-styles/.

Familiar, B. [bobfamiliar]. (2009, septiembre 21). Using LAAAM to Make Good Architectural Decisions, Fast! [Archivo de video]. Recuperado de [https://channel9.msdn.com/shows/ARCast.TV/ARCastTV-Using-LAAAM-to-Make-Good-Architectural-Decisions-Fast/]

Feitelson, D., Frachtenberg, E. & Beck, K. (2013). Development and deployment at Facebook. *IEEE Internet Computing, 17*(4), 8-17.

Ford, N., Parsons, R. & Kua, P. (2017), N. Ford, R. Parsons & P. Kua, *Building Evolutionary Architectures - Support Constant Change*, O'Reilly Media, 2017

Fowler, M. (2003). Who Needs an Architect?. *IEEE Software, 20*(5), 11-13.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R. & Stafford, R. (2002), *Patterns of Enterprise Application Architecture*, Addison Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns*, Addison-Wesley.

Geerling, J. (2018), *Ansible for DevOps - Server and configuration management for humans*, LeanPub, 2018

Grady, R. & Casswell, D. (1987), *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall.

Harper,K., & Zheng, J. (2015), *Exploring Software Architecture Context*, 2015 12th Working IEEE/IFIP Conference on Software Architecture. Congreso llevado a cabo en Montreal, Canadá.

Hilliard, R. (2014), *Architecture Viewpoint Template for ISO/IEC/IEEE 42010*. Recuperado de www.iso-architecture.org/42010/templates/

Humble, J. & Farley, D. (2010), *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley.

Huttermann, M. (2012), *DevOps for Developers*, Apress

IEEE (1990), Ieee, *IEEE Standard Glossary of Software Engineering Terminology*.

Ionita, M. & Hammer, D. (2002), *Scenario-based software architecture evaluation methods: An overview*, Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering. Congreso llevado a cabo en Orlando, Estados Unidos.

ISO 24765 (2010), *ISO/IEC/IEEE 24765:2010 - Systems and software engineering – Vocabulary*, Iso/Iec Ieee.

ISO 25010 (2011). (2011). Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Recuperado de https://www.iso.org/standard/35733.html

ISO/IEC 9126. (Sin fecha). En Wikipedia. Recuperado el 19 de mayo de 2018 de https://en.wikipedia.org/wiki/ISO/IEC_9126

ISO 42010 (2011), International Organization Of Standardization, *ISO/IEC/IEEE 42010:2011 - Systems and software engineering – Architecture description*, ISOIECIEEE 420102011E Revision of ISOIEC 420102007 and IEEE Std 14712000, 2011

ISO/IEC 42010. (2015). ISO, *ISO/IEC/IEEE 42010: Architecture Frameworks*.

ISO/IEC 42010. (Sin fecha). En Wikipedia. Recuperado el 19 de mayo de 2018 de https://en.wikipedia.org/wiki/ISO/IEC_42010

Javed, M. (2007), *A Rationale Focused Software Architecture Documentation method (RFSAD)*, Master thesis in software engineering and management, Department of Applied Information Technology, IT University of Göteborg, Suecia.

Jia, J., Fischer G. & Dyer, J. (1998). Attribute weighting methods and decision quality in the presence of response error: a simulation study. *Journal of Behavioral Decision Making, 11*(2), 85-105.

Josey, A. (2011), *TOGAF Version 9.1 Enterprise Edition*, The Open Group.

Kazman, R., Bass, L., Abowd G. & Webb, M. (1994), *SAAM: A Method for Analyzing the Properties of Software Architecture*, Proceedings of 16th International Conference on Software Engineering. Congreso llevado a cabo en Sorrento, Italia.

Kazman, R., Bass, L., Klein, M., Lattanze, T. & Northrop, L. (2005). A basis for analyzing software architecture analysis methods. *Software Quality Journal, 13*(4), 329-355.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. & Carriere, J. (1998) *The Architecture Tradeoff Analysis Method*, Software Engineering Institute, Carnegie Mellon University, Estados Unidos.

Kazman, R., Klein, R. & Clements, P. (2000), *ATAM: Method for Architecture Evaluation*. Carnegie Mellon University.

Kotusev, S. (2016), *Enterprise Architecture Is Not TOGAF*. British Computer Society. Recuperado el 21 de mayo de 2018 de https://www.bcs.org/content/conWebDoc/55547

Kurniawan, Y. (2016), *Ansible for AWS*, LeanPub.

List of software architecture styles and patterns. (Sin fecha). En Wikipedia. Recuperado el 19 de mayo de 2018 de https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns

Maier, M., Emery, D. & Hilliard, R. (2001) *Software Architecture: Introducing IEEE Standard 1471.*

Mattsson, M., Grahn, H. & Mårtensson, F. (2006), *Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability*, Department of Systems and Software Engineering, School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden.

Microsoft (2009). Microsoft Application Architecture Guide, 2nd Edition. Recuperado de https://msdn.microsoft.com/en-us/library/ff650706.aspx

Nygard, M. (2007), *Release It! - Design and Deploy Production-Ready Software*, The Pragmatic Programmers.

OpenScientist (2011), *Finalizing a Definition of "Citizen Science" and "Citizen Scientists"*. Recuperado el 21 de mayo de 2018 de
http://www.openscientist.org/2011/09/finalizing-definition-of-citizen.html

Pareto, L., Sandberg, A., Eriksson, P., & Ehnebom, S. (2011), *Prioritizing Architectural Concerns*, 2011 Ninth Working IEEE/IFIP Conference on Software Architecture. Congreso llevado a cabo en Boulder, Estados Unidos.

Puppet (2017). State of DevOps Report. Recuperado de https://puppet.com/resources/whitepaper/state-of-devops-report

Rasmussen, J. (2008), *Understanding Software Architectures: Tracing architectural knowledge in software architecture documentation*, University of Groningen, Holanda.

Richards, M. (2015), *Software Architecture Patterns*, O'Reilly Media.

Roberts, M. (2016), *Serverless Architectures*, https://martinfowler.com/articles/serverless.html, Accedido el 21 de mayo de 2018, 2016

Sarkar, A., & Shah, A. (2015), *Learning AWS*, Packt Publishing.

Spinellis, D. & Gousios, G. (2009), *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*, O'Reilly.

Stal, M. (2011), *Trust is good, Control is better - Software Architecture Assessment*. InfoQ. Recuperado el 19 de mayo de 2018 de https://www.infoq.com/articles/softwarearch-assessment

Tanenbaum, A. & van Steen, M. (2007), *Distributed Systems: Principles and Paradigms, 2/E*, Marteen van Steen.

Togaf 9.2 (2018). Welcome to the TOGAF® Standard, Version 9.2, a standard of The Open Group. Recuperado de http://pubs.opengroup.org/architecture/togaf9-doc/arch/

Torberntsson, K. & Rydin, Y. (2014), *A Study of Configuration Management Systems*, Universidad de Upsala, Suecia

van Heesch, U., Avgeriou, P., & Hilliard, R. (2012), *Forces on Architecture Decisions – A Viewpoint*, 2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture. Congreso llevado a cabo en Helsinki, Finlandia.

Venezia, P. (2013), *Puppet vs. Chef vs. Ansible vs. Salt*. ComputerWorld. Recuperado el 21 de mayo de 2018 de https://www.computerworld.com/article/2486206/virtualization/puppet-vs--chef-vs--ansible-vs--salt.html

Verona, J. (2016), *Practical DevOps*, Packt Publishing.

Verona, J., Duffy, M., & Swartout, P. (2016), *Learning DevOps: Continuously Deliver Better Software*, Packt Publishing.

Zooniverse. (Sin fecha). Vision From the Skies: The Plastic Tide Project. Recuperado de https://www.zooniverse.org/projects/theplastictide/the-plastic-tide/about/research

Apéndice A

Encuesta a los desarrolladores

La siguiente es una encuesta realizada a los desarrolladores de las distintas aplicaciones que van a ejecutarse en el contexto de Cientópolis. El objetivo es conocer las demandas tecnológicas de cada aplicación. En este anexo se reproduce la encuesta, mientras que las respuestas y un análisis de las mismas fueron trabajadas en el Capítulo 3.

A.0.1. Características generales del servidor donde corre la aplicación

Indicar, para el ambiente de desarrollo[1]:

- Sistema operativo
- Memoria RAM
- Espacio en disco
- Cores de CPU

A.0.2. Sobre las tecnologías utilizadas

- Servidor web
- Servidor de base de datos (PostgreSQL, Mongo, etc)
- Otros servicios corriendo (Memcache, RabbitMQ, Postfix, etc)
- ¿Cómo instalaste esos servicios? ¿A mano, con alguna herramienta de automatización, etc?

[1] No existía, al momento de la encuesta, distinción entre ambientes de desarrollo, *testing* y producción.

- Lenguajes de programación usados
- Frameworks y librerías usadas
- Otras tecnologías usadas en el desarrollo de la aplicación (Flash, Websocket, etc). ¿Estás usando alguna herramienta, librería o protocolo que esté en fase "experimental"?

A.0.3. Sobre la aplicación en sí

- Método de *deployment* de la aplicación (A mano, usando alguna herramienta de automatización, etc). ¿Cómo subís el código nuevo y actualizás las bases de datos?
- ¿El *deployment* de la aplicación implica que tenés que loguearte al servidor y ejecutar tareas o scripts?
- ¿El usuario mantiene sesión? ¿La aplicación soporta *Single Sign On* (SSO)?
- ¿Existen distintos tipos de usuario (Admin, usuario común, anónimo, etc)?
- ¿Cómo se loguea el usuario? ¿A través de Facebook, son usuarios locales guardados en la base de datos, usando una API de Google, usuarios en un LDAP?
- ¿La aplicación está diseñada como un backend y un frontend que se comunican entre sí? ¿Es una aplicación monolítica? ¿Está pensada como microservicios?
- ¿La aplicación está diseñada considerando que delante puede haber una cache o un proxy reverso?
- ¿Cómo se actualizan la aplicación y las librerías? (A mano, corriendo algún script, etc)
- ¿Dónde estás desarrollando y testeando la aplicación? ¿En tu misma máquina, en máquinas virtuales?
- ¿La aplicación almacena algún tipo de logs? (No me refiero a los logs del servidor o del framework, sino que explícitamente la aplicación loguee algo)
- ¿La aplicación mantiene estadísticas y logs? Si lo hace, ¿es en la misma base de datos de la aplicación o los manda a un servidor externo?

A.0.4. Sobre la documentación y el código

- ¿Qué editor o IDE usás?
- Sistema operativo base sobre el que desarrollás (no donde corre la aplicación, sino donde escribís el código)
- ¿Estás versionando el código en algún lado?

- ¿Tenés un código de estilos para la documentación, el manual de usuario, la escritura de logs, la ayuda y el código de la aplicación?
- ¿Qué tecnología de virtualización usás para el proyecto de Cientópolis? (VMWare, Proxmox, Docker, por ejemplo)
- ¿Están bien marcados los ambientes de desarrollo, testing y producción?

Apéndice B

Utility Tree y Escenarios

B.1. Utility Tree

Este anexo muestra el *Utility tree* completo para la arquitectura de Cientópolis. Para que sea más sencillo su lectura, se divide en varias figuras. La Figura B.1 muestra, en general, como se decidió agrupar las propiedades evaluadas. En dicha figura no se muestran los escenarios considerados.

La Figura B.2 muestra los escenarios relacionados con la usabilidad y las propiedades bajo ella agrupadas. De similar manera se hace con la Figura B.3 para los escenarios de seguridad, la Figura B.4 para la mantenibilidad y la Figura B.5 para la performance,

B.2. Escenarios

En esta sección se muestran los cuadros con todos los escenarios para los atributos de usabilidad, seguridad, mantenibilidad y performance. El Cuadro B.1 muestra los escenarios de usabilidad.

En el Cuadro B.2 se muestran los escenarios de seguridad planteados por los *stakeholders* de Cientópolis.

El Cuadro B.3 muestra los escenarios planteados por los *stakeholders* para el atributo mantenibilidad.

El Cuadro B.4 muestra los escenarios elicitados para la performance. Para mayor legibilidad se eligió mostrar los escenarios pertenecientes a tiempo de recuperación y tiempo entre fallos como directamente descendientes del atributo disponibilidad, de ahí que el peso de esos escenarios podría parecer incorrecto a primera vista.

Por último, el Cuadro B.5 muestra todos los escenarios según su peso.

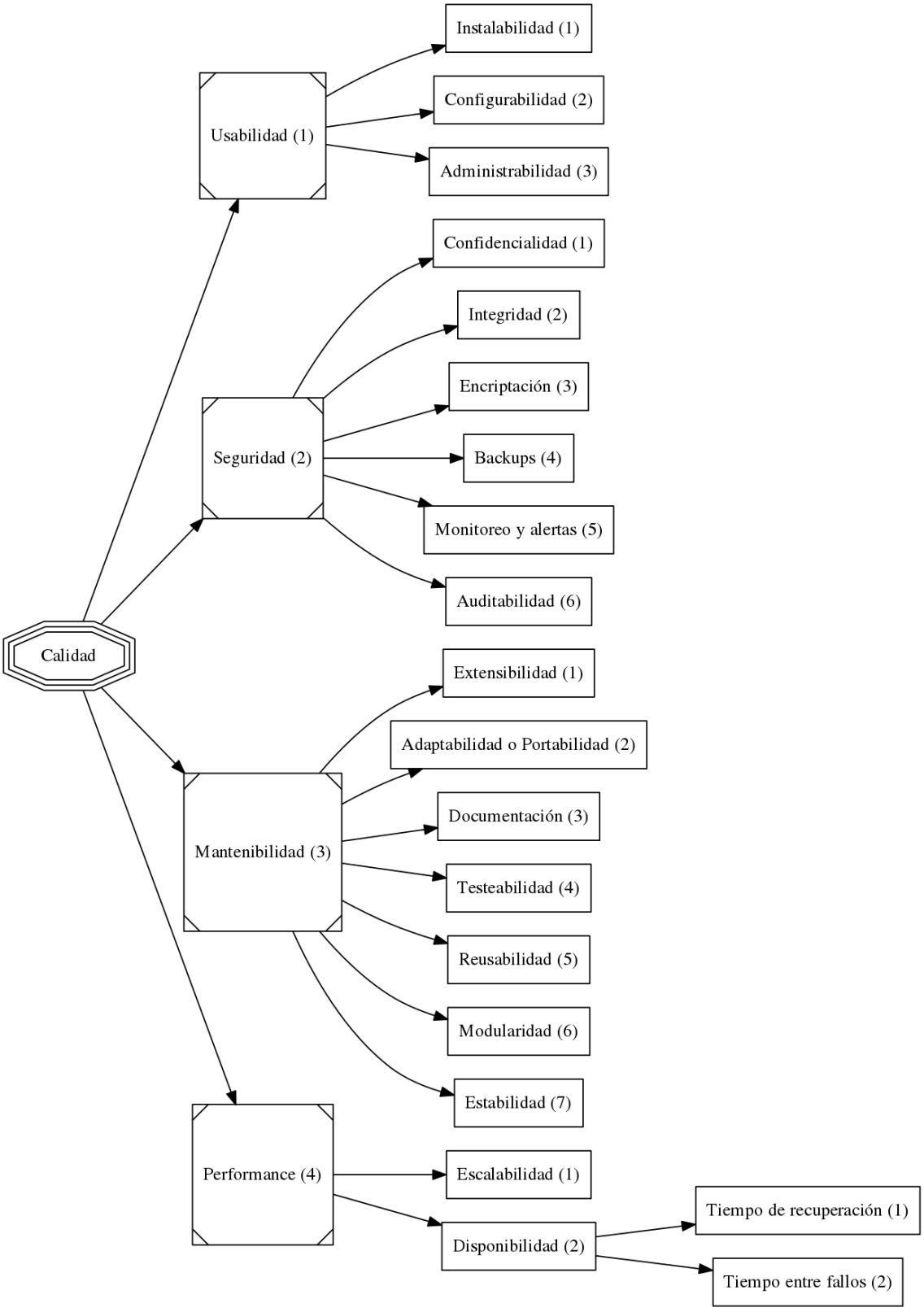

Figura B.1: Utility tree general

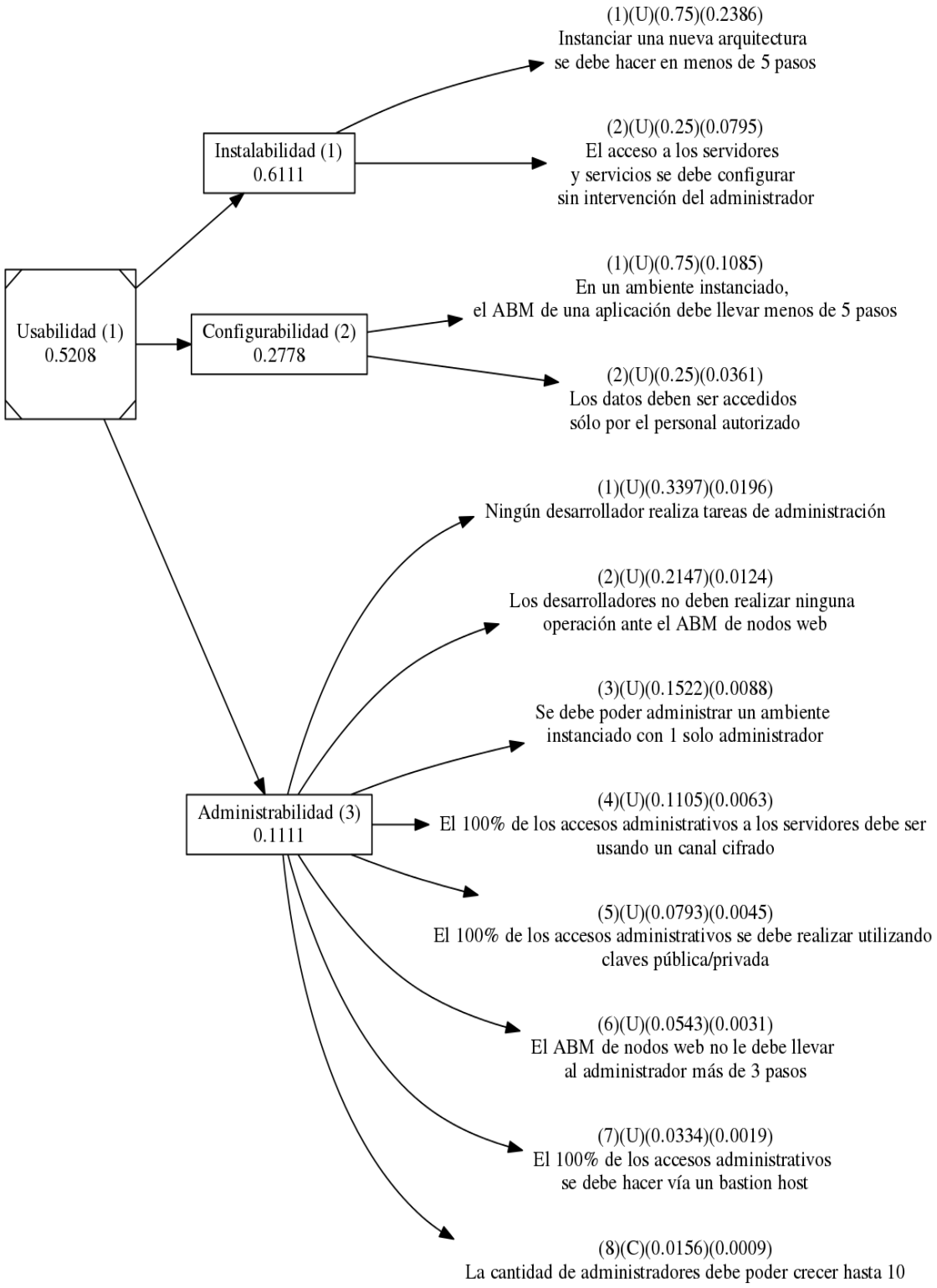

Figura B.2: *Utility tree* y escenarios de Usabilidad

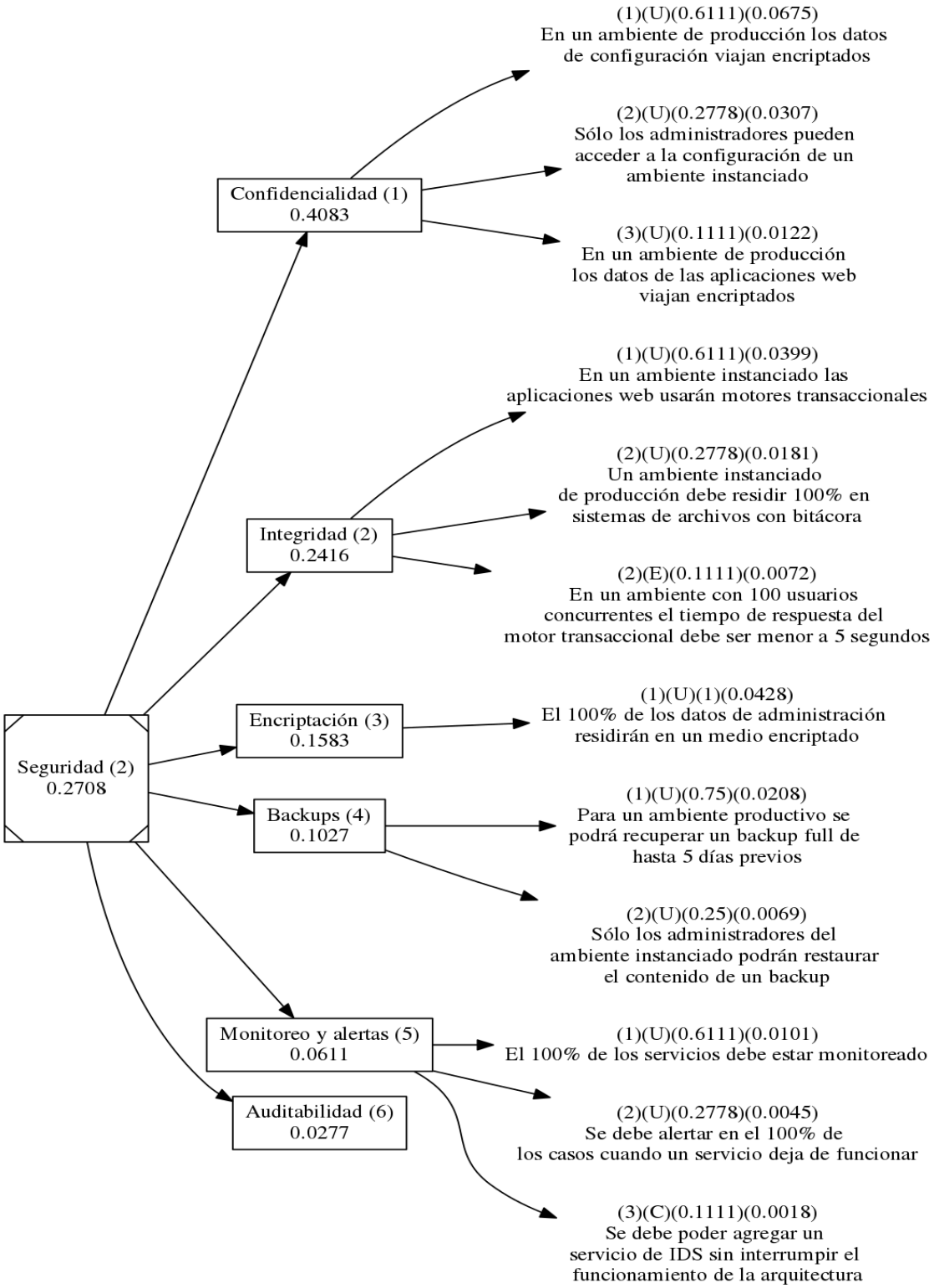

Figura B.3: *Utility tree* y escenarios de Seguridad

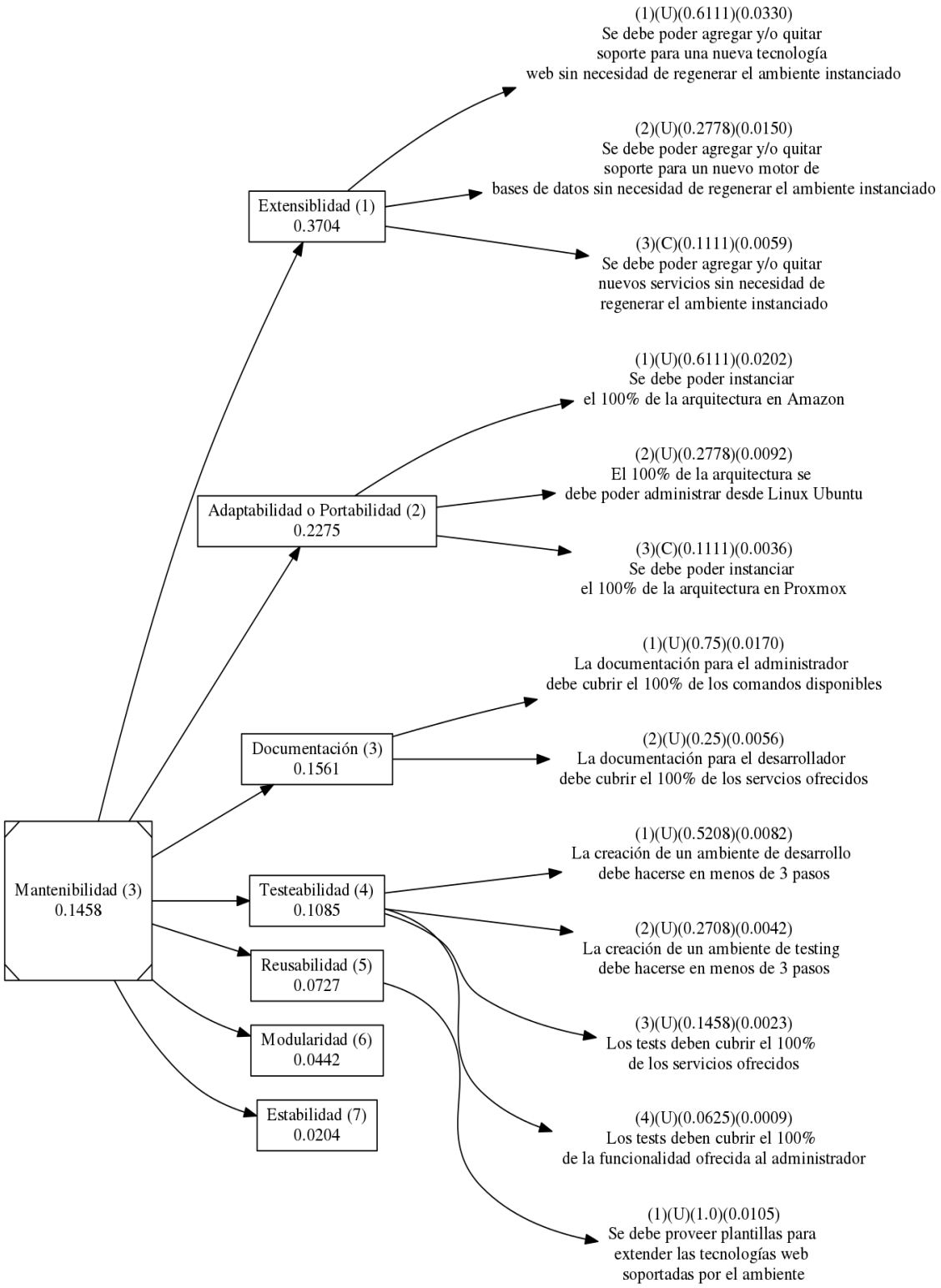

Figura B.4: *Utility tree* y escenarios de Mantenibilidad

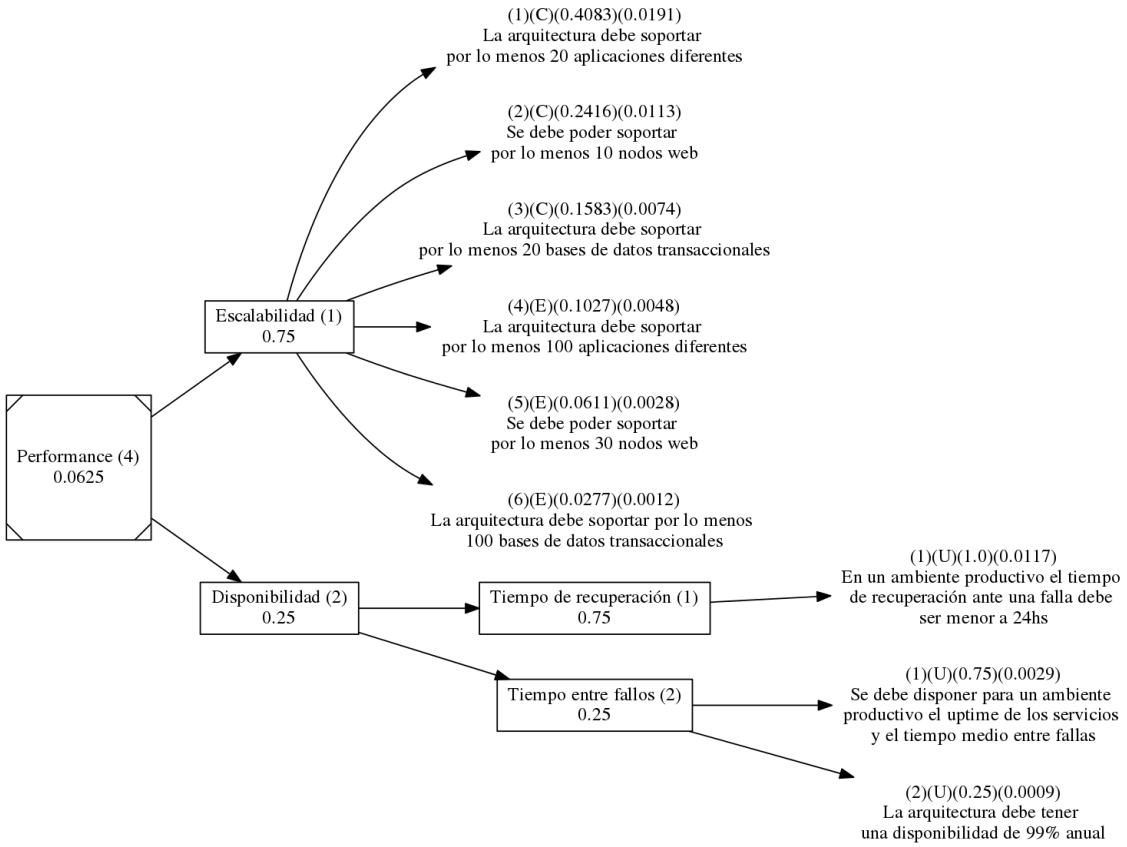

Figura B.5: *Utility tree* y escenarios de Performance

Usabilidad					
Núm.	Atributo	Prioridad	Tipo	Peso	Escenario
1	Instalabilidad	1	U	0.2386	Instanciar una nueva arquitectura se debe hacer en menos de 5 pasos.
2	Instalabilidad	2	U	0.0795	El acceso a los servidores y servicios se debe configurar sin intervención del administrador.
3	Configurabilidad	1	U	0.1085	En un ambiente instanciado, el ABM de una aplicación debe llevar menos de 5 pasos.
4	Configurabilidad	2	U	0.0361	Los datos deben ser accedidos sólo por el personal autorizado.
5	Administrabilidad	1	U	0.0196	Ningún desarrollador realiza tareas de administración.
6	Administrabilidad	2	U	0.0124	Los desarrolladores no deben realizar ninguna operación ante el ABM de nodos web.
7	Administrabilidad	3	U	0.0088	Se debe poder administrar un ambiente instanciado con 1 solo administrador.
8	Administrabilidad	4	U	0.0063	El 100 % de los accesos administrativos a los servidores debe ser usando un canal cifrado.
9	Administrabilidad	5	U	0.0045	El 100 % de los accesos administrativos se debe realizar utilizando claves pública/privada.
10	Administrabilidad	6	U	0.0031	El ABM de nodos web no le debe llevar al administrador más de 3 pasos.
11	Administrabilidad	7	U	0.0019	El 100 % de los accesos administrativos se debe hacer vía un bastion host.
12	Administrabilidad	8	C	0.0009	La cantidad de administradores debe poder crecer hasta 10.

Cuadro B.1: cuadro con los escenarios de usabilidad elicitados para Cientópolis.

Seguridad					
Núm.	Atributo	Prioridad	Tipo	Peso	Escenario
13	Confidencialidad	1	U	0.0675	En un ambiente de producción los datos de configuración viajan encriptados.
14	Confidencialidad	2	U	0.0307	Sólo los administradores pueden acceder a la configuración de un ambiente instanciado.
15	Confidencialidad	3	U	0.0122	En un ambiente de producción los datos de las aplicaciones web viajan encriptados.
16	Integridad	1	U	0.0399	En un ambiente instanciado las aplicaciones web usarán motores transaccionales.
17	Integridad	2	U	0.0181	Un ambiente instanciado de producción debe residir 100 % en sistemas de archivos con bitácora.
18	Integridad	3	E	0.0072	En un ambiente con 100 usuarios concurrentes el tiempo de respuesta del motor transaccional debe ser menor a 5 segundos.
19	Encriptación	1	U	0.0428	El 100 % de los datos de administración residirán en un medio encriptado.
20	Backups	1	U	0.0208	Para un ambiente productivo se podrá recuperar un backup full de hasta 5 días previos.
21	Backups	2	U	0.0069	Sólo los administradores del ambiente instanciado podrán restaurar el contenido de un backup.
22	Monitoreo y alertas	1	U	0.0101	El 100 % de los servicios debe estar monitoreado.
23	Monitoreo y alertas	2	U	0.0045	Se debe alertar en el 100 % de los casos cuando un servicio deja de funcionar.
24	Monitoreo y alertas	3	C	0.0018	Se debe poder agregar un servicio de IDS sin interrumpir el funcionamiento de la arquitectura.

Cuadro B.2: cuadro con los escenarios de seguridad elicitados para Cientópolis.

Mantenibilidad					
Núm.	Atributo	Prioridad	Tipo	Peso	Escenario
25	Extensibilidad	1	U	0.0330	Se debe poder agregar y/o quitar soporte para una nueva tecnología web sin necesidad de regenerar el ambiente instanciado.
26	Extensibilidad	2	U	0.0150	Se debe poder agregar y/o quitar soporte para un nuevo motor de bases de datos sin necesidad de regenerar el ambiente instanciado.
27	Extensibilidad	3	C	0.0059	Se debe poder agregar y/o quitar nuevos servicios sin necesidad de regenerar el ambiente instanciado.
28	Adaptabilidad	1	U	0.0202	Se debe poder instanciar el 100 % de la arquitectura en Amazon.
29	Adaptabilidad	2	U	0.0092	El 100 % de la arquitectura se debe poder administrar desde Linux Ubuntu.
30	Adaptabilidad	3	C	0.0036	Se debe poder instanciar el 100 % de la arquitectura en Proxmox.
31	Documentación	1	U	0.0170	La documentación para el administrador debe cubrir el 100 % de los comandos disponibles.
32	Documentación	2	U	0.0056	La documentación para el desarrollador debe cubrir el 100 % de los servcios ofrecidos.
33	Testeabilidad	1	U	0.0082	La creación de un ambiente de desarrollo debe hacerse en menos de 3 pasos.
34	Testeabilidad	2	U	0.0042	La creación de un ambiente de testing debe hacerse en menos de 3 pasos.
35	Testeabilidad	3	U	0.0023	Los tests deben cubrir el 100 % de los servicios ofrecidos.
36	Testeabilidad	4	U	0.0009	Los tests deben cubrir el 100 % de la funcionalidad ofrecida al administrador.
37	Reusabilidad	1	U	0.0105	Se debe proveer plantillas para extender las tecnologías web soportadas por el ambiente.

Cuadro B.3: cuadro con los escenarios elicitados relacionados con la mantenibilidad de la arquitectura de Cientópolis.

Performance					
Núm.	Atributo	Prioridad	Tipo	Peso	Escenario
38	Escalabilidad	1	C	0.0191	La arquitectura debe soportar por lo menos 20 aplicaciones diferentes.
39	Escalabilidad	2	C	0.0113	Se debe poder soportar por lo menos 10 nodos web.
40	Escalabilidad	3	C	0.0074	La arquitectura debe soportar por lo menos 20 bases de datos transaccionales.
41	Escalabilidad	4	E	0.0048	La arquitectura debe soportar por lo menos 100 aplicaciones diferentes.
42	Escalabilidad	5	E	0.0028	Se debe poder soportar por lo menos 30 nodos web.
43	Escalabilidad	6	E	0.0012	La arquitectura debe soportar por lo menos 100 bases de datos transaccionales.
44	Disponibilidad	1	U	0.0117	En un ambiente productivo el tiempo de recuperación ante una falla debe ser menor a 24hs.
45	Disponibilidad	2	U	0.0029	Se debe disponer para un ambiente productivo el uptime de los servicios y el tiempo medio entre fallas.
46	Disponibilidad	3	U	0.0009	La arquitectura debe tener una disponibilidad de 99 % anual.

Cuadro B.4: cuadro con los escenarios elicitados para la performance.

Núm.	Atributo	Prioridad	Tipo	Peso	Escenario
1	Instalabilidad	1	U	0.2386	Instanciar una nueva arquitectura se debe hacer en menos de 5 pasos.
3	Configurabilidad	1	U	0.1085	En un ambiente instanciado, el ABM de una aplicación debe llevar menos de 5 pasos.
2	Instalabilidad	2	U	0.0795	El acceso a los servidores y servicios se debe configurar sin intervención del administrador.
13	Confidencialidad	1	U	0.0675	En un ambiente de producción los datos de configuración viajan encriptados.
19	Encriptación	1	U	0.0428	El 100 % de los datos de administración residirán en un medio encriptado.
16	Integridad	1	U	0.0399	En un ambiente instanciado las aplicaciones web usarán motores transaccionales.
4	Configurab.	2	U	0.0361	Los datos deben ser accedidos sólo por el personal autorizado.
25	Extensibilidad	1	U	0.0330	Se debe poder agregar y/o quitar soporte para una nueva tecnología web sin necesidad de regenerar el ambiente instanciado.
14	Confidencialidad	2	U	0.0307	Sólo los administradores pueden acceder a la configuración de un ambiente instanciado.
20	Backups	1	U	0.0208	Para un ambiente productivo se podrá recuperar un backup full de hasta 5 días previos.
28	Adaptabilidad	1	U	0.0202	Se debe poder instanciar el 100 % de la arquitectura en Amazon.
5	Administrabilidad	1	U	0.0196	Ningún desarrollador realiza tareas de administración.
38	Escalabilidad	1	C	0.0191	La arquitectura debe soportar por lo menos 20 aplicaciones diferentes.
17	Integridad	2	U	0.0181	Un ambiente instanciado de producción debe residir 100 % en sistemas de archivos con bitácora.

31	Documentación	1	U	0.0170	La documentación para el administrador debe cubrir el 100 % de los comandos disponibles.
26	Extensibilidad	2	U	0.0150	Se debe poder agregar y/o quitar soporte para un nuevo motor de bases de datos sin necesidad de regenerar el ambiente instanciado.
6	Administrabilidad	2	U	0.0124	Los desarrolladores no deben realizar ninguna operación ante el ABM de nodos web.
15	Confidencialidad	3	U	0.0122	En un ambiente de producción los datos de las aplicaciones web viajan encriptados.
44	Disponibilidad	1	U	0.0117	En un ambiente productivo el tiempo de recuperación ante una falla debe ser menor a 24hs.
39	Escalabilidad	2	C	0.0113	Se debe poder soportar por lo menos 10 nodos web.
37	Reusabilidad	1	U	0.0105	Se debe proveer plantillas para extender las tecnologías web soportadas por el ambiente.
22	Monitoreo y alertas	1	U	0.0101	El 100 % de los servicios debe estar monitoreado.
29	Adaptabilidad	2	U	0.0092	El 100 % de la arquitectura se debe poder administrar desde Ubuntu Linux.
7	Administrabilidad	3	U	0.0088	Se debe poder administrar un ambiente instanciado con 1 solo administrador.
33	Testeabilidad	1	U	0.0082	La creación de un ambiente de desarrollo debe hacerse en menos de 3 pasos.
40	Escalabilidad	3	C	0.0074	La arquitectura debe soportar por lo menos 20 bases de datos transaccionales.
18	Integridad	3	E	0.0072	En un ambiente con 100 usuarios concurrentes el tiempo de respuesta del motor transaccional debe ser menor a 5 segundos.

21	Backups	2	U	0.0069	Sólo los administradores del ambiente instanciado podrán restaurar el contenido de un backup.
8	Administrabilidad	4	U	0.0063	El 100 % de los accesos administrativos a los servidores debe ser usando un canal cifrado.
27	Extensibilidad	3	C	0.0059	Se debe poder agregar y/o quitar nuevos servicios sin necesidad de regenerar el ambiente instanciado.
32	Documentación	2	U	0.0056	La documentación para el desarrollador debe cubrir el 100 % de los servcios ofrecidos.
41	Escalabilidad	4	E	0.0048	La arquitectura debe soportar por lo menos 100 aplicaciones diferentes.
9	Administrabilidad	5	U	0.0045	El 100 % de los accesos administrativos se debe realizar utilizando claves pública/privada.
23	Monitoreo y alertas	2	U	0.0045	Se debe alertar en el 100 % de los casos cuando un servicio deja de funcionar.
34	Testeabilidad	2	U	0.0042	La creación de un ambiente de testing debe hacerse en menos de 3 pasos.
30	Adaptabilidad	3	C	0.0036	Se debe poder instanciar el 100 % de la arquitectura en Proxmox.
10	Administrabilidad	6	U	0.0031	El ABM de nodos web no le debe llevar al administrador más de 3 pasos.
45	Disponibilidad	2	U	0.0029	Se debe disponer para un ambiente productivo el uptime de los servicios y el tiempo medio entre fallas.
42	Escalabilidad	5	E	0.0028	Se debe poder soportar por lo menos 30 nodos web.
35	Testeabilidad	3	U	0.0023	Los tests deben cubrir el 100 % de los servicios ofrecidos.
11	Administrabilidad	7	U	0.0019	El 100 % de los accesos administrativos se debe hacer vía un bastion host.
24	Monitoreo y alertas	3	C	0.0018	Se debe poder agregar un servicio de IDS sin interrumpir el funcionamiento de la arquitectura.

43	Escalabilidad	6	E	0.0012	La arquitectura debe soportar por lo menos 100 bases de datos transaccionales.
12	Administrabilidad	8	C	0.0009	La cantidad de administradores debe poder crecer hasta 10.
36	Testeabilidad	4	U	0.0009	Los tests deben cubrir el 100 % de la funcionalidad ofrecida al administrador.
46	Disponibilidad	3	U	0.0009	La arquitectura debe tener una disponibilidad de 99 % anual.

Cuadro B.5: cuadro con los escenarios ordenados según su peso.

Núm.	Peso	Alternativa 1	Alternativa 2	Alternativa 3
1	0.2386	Muy Bien	Muy Bien	Muy Bien
3	0.1085	Muy Bien	Muy Bien	Muy Bien
2	0.0795	Muy Bien	Muy Bien	Muy Bien
13	0.0675	Muy Bien	Muy Bien	Muy Bien
19	0.0428	Muy Bien	Muy Bien	Muy Bien
16	0.0399	Muy Bien	Muy Bien	Muy Bien
4	0.0361	Muy Bien	Muy Bien	Muy Bien
25	0.0330	Mal	Bien	Bien
14	0.0307	Muy Bien	Muy Bien	Muy Bien
20	0.0208	Muy Bien	Muy Bien	Muy Bien
28	0.0202	Muy Bien	Muy Bien	Muy Bien
5	0.0196	Muy Bien	Muy Bien	Muy Bien
38	0.0191	Muy Bien	Muy Bien	Muy Bien
17	0.0181	Bien	Bien	Bien
31	0.0170	Muy Bien	Muy Bien	Muy Bien
26	0.0150	Muy Bien	Muy Bien	Muy Bien
6	0.0124	Bien	Muy Bien	Bien
15	0.0122	Muy Bien	Muy Bien	Mal
44	0.0117	Muy Bien	Muy Bien	Muy Bien
39	0.0113	Bien	Muy Bien	Muy Bien
37	0.0105	Bien	Muy Bien	Muy Bien
22	0.0101	Muy Bien	Muy Bien	Muy Bien
29	0.0092	Muy Bien	Muy Bien	Muy Bien
7	0.0088	Muy Bien	Muy Bien	Bien
33	0.0082	Muy Bien	Muy Bien	Mal
40	0.0074	Muy Bien	Muy Bien	Muy Bien
18	0.0072	Bien	Bien	Bien
21	0.0069	Muy Bien	Muy Bien	Muy Bien
8	0.0063	Muy Bien	Muy Bien	Muy Bien
27	0.0059	Bien	Bien	Bien
32	0.0056	Muy Bien	Muy Bien	Muy Bien
41	0.0048	Muy Bien	Muy Bien	Muy Bien
9	0.0045	Muy Bien	Muy Bien	Muy Bien
23	0.0045	Muy Bien	Muy Bien	Muy Bien
34	0.0042	Muy Bien	Muy Bien	Mal
30	0.0036	Muy Bien	Muy Bien	Mal

10	0.0031	Muy Bien	Muy Bien	Muy Bien
45	0.0029	Muy Bien	Muy Bien	Muy Bien
42	0.0028	Bien	Muy Bien	Muy Bien
35	0.0023	Muy Bien	Muy Bien	Mal
11	0.0019	Muy Bien	Muy Bien	Bien
24	0.0018	Muy Bien	Muy Bien	Muy Bien
43	0.0012	Muy Bien	Muy Bien	Muy Bien
12	0.0009	Muy Bien	Muy Bien	Muy Bien
36	0.0009	Muy Bien	Muy Bien	Mal
46	0.0009	Muy Bien	Muy Bien	Muy Bien
Total	1.0[1]	1.8266	1.8966	1.8107

Cuadro B.6: cuadro con los escenarios y cómo se ajusta cada alternativa.

El Cuadro B.6 muestra cómo se ajusta cada alternativa a los escenarios. Los valores elegidos para mostrar el nivel de ajuste son: **Mal (0)**, **Bien (1)** y **Muy Bien (2)**.

[1] El valor es aproximado

Apéndice C

Documentación de la arquitectura

C.1. Stakeholders

Los *stakeholders* son personas, grupos u organizaciones con algún tipo de interés o *concern* en el sistema. En esta sección se describen los *stakeholders* de Cientópolis y su interacción con la arquitectura.

Además, el Cuadro C.1 indica para cada *stakeholder* los niveles de interés en la realización del proyecto, participación activa en el proyecto e influencia. Esta categorización se refiere exclusivamente a la arquitectura de Cientópolis.

C.1.1. Directores de Cientópolis

Son las personas que dirigen el proyecto de Cientópolis. Interactúan con la arquitectura en varios niveles:

- Se encargan de brindan apoyo económico en la forma de tiempo de cómputo en Amazon y el clúster de Proxmox propio.

Stakeholder	Nivel de interés	Nivel de participación	Nivel de influencia
Directores de Cientópolis	Muy Alto	Medio	Media
Desarrolladores de aplicaciones	Alto	Bajo	Media
Administradores de Cientópolis	Alto	Medio	Media
Usuarios de las aplicaciones	Bajo	Bajo	Bajo
Administradores de la infraestructura	Medio	Bajo	Medio
Arquitectos de Cientópolis	Muy Alto	Muy Alto	Muy alto

Cuadro C.1: cuadro con los *stakeholders* de Cientópolis.

- Dictan la mayoría de los requerimientos funcionales y no funcionales de la arquitectura.
- Son el nexo principal con los demás *stakeholders*.

C.1.2. Desarrolladores de aplicaciones

Los desarrolladores son las personas que programan las aplicaciones que se ejecutarán sobre la arquitectura. Estas aplicaciones son la materialización de un proyecto de investigación y serán utilizadas por los usuarios finales. Los desarrolladores interactúan con la arquitectura de las siguientes formas:

- Las aplicaciones que programan consumen servicios brindados por la arquitectura, por lo tanto, dictan requerimientos funcionales y no funcionales.
- Deben subir las aplicaciones y actualizaciones a la arquitectura, por lo tanto, utilizan la arquitectura como usuarios finales que hacen *login*, realizan diversas tareas y hacen *logout*.

C.1.3. Administradores de Cientópolis

Los administradores de Cientópolis son las personas encargadas de operar la arquitectura. Interactúan con ella de diversas maneras:

- Realizan el ABM[1] de las aplicaciones de Cientópolis, así como de los usuarios y bases de datos asociados.
- Se encargan del monitoreo de la arquitectura.
- Son los encargados de restaurar la arquitectura y sus datos en caso de fallo.
- Se encargan de escalar el sistema.

Notar que tanto el monitoreo como los backups pueden verse también como elementos de la infraestructura. Sin embargo, en este caso, se asume que la arquitectura de Cientópolis está aislada y, por lo tanto, se deben proveer aquellos elementos de infraestructura.

C.1.4. Usuarios de las aplicaciones

Este es el grupo más heterogéneo y se refiere a todas aquellas personas que utilizan las aplicaciones alojadas en Cientópolis. Estos usuarios se ve afectados principalmente por cuestiones no funcionales, como son los tiempos de respuesta o el grado de disponibilidad de las aplicaciones. Pero también asumen detalles funcionales, como la privacidad de sus datos.

[1]Siglas que hacen referencia a la alta, baja y modificación de, en este caso, aplicaciones de Cientópolis

C.1.5. Administradores de la infraestructura

Son los encargados de administrar los elementos propios de la infraestructura sobre la que se apoya Cientópolis, como son los virtualizadores o el suministro de energía. No se espera que interactúen directamente con la arquitectura de Cientópolis, pero sí que lo hagan con los administradores de Cientópolis.

C.1.6. Arquitectos de Cientópolis

Son las personas encargadas de diseñar y desarrollar la arquitectura. Este grupo interactúa de varias maneras con la arquitectura:

- Como se dijo anteriormente, se encarga del diseño y desarrollo.
- También se encarga del testing y documentación.
- Se encarga de la transferencia de conocimiento y *training* de los administradores de Cientópolis.
- Como desarrolladores, realizan cambios y actualizaciones sobre el código de la arquitectura, como puede ser soportar una nueva tecnología. Es decir, de mantener y extender el sistema.

C.2. Concerns

Los *concerns* son aquellos intereses que tienen los *stakeholders* en la arquitectura. Es decir, son aquellas inquietudes que cada *stakeholder* manifiesta acerca de la arquitectura.

Instalabilidad: se espera que el proceso de instalar o instanciar la arquitectura sea relativamente fácil y rápido. Es decir, que no se requiera una conocimiento profundo sobre todas las tecnologías involucradas y que no consuma demasiado tiempo.

Configurabilidad: por otra parte, la configuración de la arquitectura y del espacio para las aplicaciones debe ser lo más sencilla posible.

Administrabilidad: de la misma manera, la operatoria de la arquitectura no debe demandar demasiado tiempo.

Confidencialidad: se espera que los datos de configuración y de las aplicaciones viajen encriptados.

Integridad: también se espera que la arquitectura colabore en mantener la integridad de los datos.

Backups: para una rápida recuperación se espera que los backups tengan una cobertura de por lo menos 5 días y sea fácil su restauración.

Concern	Directores de Cientópolis	Desarrolladores de aplicaciones	Administradores de Cientópolis	Usuarios de las aplicaciones	Administradores de la infraestructura	Arquitectos de Cientópolis
Instalabilidad	✓		✓			✓
Configurabilidad	✓		✓			✓
Administrabilidad	✓		✓			✓
Confidencialidad				✓		
Integridad				✓		
Backup			✓		✓	✓
Monitoreo y alertas			✓		✓	✓
Extensibilidad						✓
Documentación		✓	✓		✓	✓
Escalabilidad			✓		✓	✓
Disponibilidad			✓	✓	✓	

Cuadro C.2: cuadro con los *concerns* que le incumben a cada *stakeholder* de Cientópolis.

Monitoreo y alertas: se espera que los problemas en la arquitectura se detecten rápidamente, en lo posible antes que los usuarios de las aplicaciones tengan oportunidad de experimentarlos, por lo que se desea un sistema de monitoreo y alertas sencillo y operativo.

Extensibilidad: otra preocupación relacionada con el medio o largo plazo de Cientópolis es la fácil inclusión de nueva funcionalidad en la arquitectura y el soporte de nuevas tecnologías.

Documentación: la documentación es primordial para el correcto uso de la arquitectura por parte de los administradores y desarrolladores.

Escalabilidad: se espera que la arquitectura escale fácilmente.

Disponibilidad: se espera que la disponibilidad de la arquitectura y de las aplicaciones sea del al menos el 99 % anual.

El Cuadro C.2 muestra, para cada *stakeholder*, los *concerns* que le incumben.

Apéndice D

Glosario

Amazon web services (AWS): provee diversos servicios de computación en la nube bajo demanda, como son máquinas virtuales, balanceadores de carga, motores de bases de datos relacionales, entre otros. La interacción puede ser mediante un sitio web o usando una consola de comandos. La facturación depende de la elección del hardware, sistema operativo, grado de disponibilidad, redundancia, cantidad de IPs públicas, etc.

Amazon availability zones (AZ): Amazon divide su nube en regiones, donde cada una a su vez está dividida en varias zonas de disponibilidad (*availability zones* en inglés). Se podría pensar a cada AZ como un centro de datos (*data center* en inglés) físico y a una región como el grupo de todos esos *data centers* físicos (o AZ) que la componen. Los AZ de una región están conectados entre sí.

Amazon machine image (AMI): provee la información necesaria para ejecutar una instancia, que es una máquina virtual. Se pueden ejecutar varias instancias de una misma AMI y se pueden usar AMIs propias o públicas.

Amazon region: es un área geográfica determinada en donde existen diversas y aisladas zonas de disponibilidad (AZ). Por ejemplo, *ap-northeast-1* es la región Asía Pacífico (Tokio). Las regiones no están conectadas entre sí.

Amazon security groups: los security groups actúan como un *firewall*, permitiendo o denegando determinado tipo de tráfico. Varias instancias pueden usar el mismo *security group*.

Amazon virtual private cloud (VPC): permite ubicar los recursos de Amazon en una red virtual de administración propia. Se tiene un completo control sobre las redes definidas. Por ejemplo, se pueden configurar diferentes tablas de ruteo, *gateways*, subredes. Se puede decir que un VPC es como un *data center* virtual.

Ansible: es la herramienta de automatización elegida para este trabajo. Es una aplicación *open source* que sirve para automatizar diversas tareas, como son el aprovisionamiento de servidores, el *deployment* de aplicaciones y el manejo de la configuración. Está escrita en Python y la comunicación con los servidores administrados se realiza sobre SSH.

Ansible inventory: es un archivo de texto en donde se listan servidores por nombre o por IP. En el inventario pueden existir diversos grupos de servidores y un servidor puede estar en varios grupos. Por ejemplo, se puede tener un grupo llamado "webservers" e indicarle a Ansible que ejecute determinadas tareas solamente en los hosts del inventario que se encuentran dentro del grupo "webservers".

Ansible module: los módulos de Ansible sirven para controlar diferentes recursos. Por ejemplo, el módulo **user** se puede usar para crear, modificar o borrar usuarios del sistema. El módulo **mysql_db** sirve para agregar o quitar una base de datos MySQL en un host remoto. Al momento de escribir estas lineas existen más de 1800 módulos en la documentación oficial[1].

Ansible playbooks: sirven para definir los pasos que componen un procedimiento determinado. Ansible llama a cada uno de estos pasos *tasks* o tareas.

Ansible role: es un mecanismo para desagrupar los elementos de un *playbook* a la vez que se facilita el reuso de código. Los roles permiten cargar automáticamente archivos de variables, tareas y otras componentes más dispuestas en una estructura de archivos bien definida. Los roles se pueden compartir fácilmente y Ansible provee el sitio Ansible Galaxy[2] para tal fin.

Ansible tasks: son las tareas o pasos que componen un *playbook*. Por ejemplo, el *playbook* **configuracion_inicial.yml** podría tener las siguientes tareas: **Copiar archivo /etc/resolv.conf**, **Levantar el firewall**, **Quitar el acceso de root**. Y cada una de estas tareas se implementarían usando un módulo de Ansible.

Ansible templates: son archivos de texto en formato jinja2[3] que sirven de plantillas. Por ejemplo, se puede tener la plantilla **apache.conf.j2** que hace referencia a la variable **my_domain**. Cuando esa plantilla se copie a un servidor web, el valor que tenga **my_domain** dependerá de lo que le haya asignado el usuario.

Archivo authorized_keys: es un archivo que especifica las claves SSH que pueden usarse para hacer login con una cuenta de usuario determinada. En este archivo se encuentran las claves públicas, mientras que los usuarios tienen las correspondientes claves privadas. Ver **Claves pública/privada**.

Archivo resolv.conf: es un archivo donde se especifica los servidores de nombre que se van a utilizar.

[1] https://docs.ansible.com/ansible/latest/modules/list_of_all_modules.html
[2] https://galaxy.ansible.com/
[3] http://jinja.pocoo.org/

Backups: también llamadas copias de seguridad. Es un procedimiento mediante el cual se copian diversos datos a un servidor de resguardo con el fin de restaurarlos ante un evento de pérdida de datos.

Bacula: es una aplicación *open source* que sirve para realizar *backups*. Trabaja en modo cliente-servidor y requiere que cada servidor que se va a resguardar ejecute un agente de Bacula.

Bastion host: es un host cuyo propósito es proporcionar acceso a una red privada desde una red externa. Típicamente, luego de conectarse al *bastion host*, el usuario "salta" hacia otro servidor que está del lado privado de la red.

Bit rot: se refiere a la degradación que sufren los sistemas a medida que pasa el tiempo. Habitualmente ocurre porque no existen guías a la hora de hacer los cambios. No es un fenómeno físico, en el sentido de que el software en sí no cambia, sino que queda desactualizado con respecto al ambiente cambiante en el que reside.

Caché: es una componente de hardware o de software que almacena datos utilizados habitualmente o cuya generación es muy costosa en términos computacionales, con la esperanza de que los accesos futuros se resuelvan más rápidamente. Desde un punto de vista tecnológico la caché de una CPU y de una página web son diferentes, pero conceptualmente comparte el mismo principio, el de acelerar el acceso a la información.

Caché hit: cuando se busca un dato en la caché y éste se encuentra allí se dice que hay un *hit*.

Caché miss: cuando se busca un dato en la caché y éste no se encuentra allí se dice que hay un *miss*. En este caso, cuando el dato se recupera de su lugar original, se lo agrega en la caché con la esperanza de acelerar sus futuros accesos.

Claves pública/privada: en la criptografía asimétrica el usuario dispone de un par de claves, una pública y otra privada. Cualquier persona puede conocer la clave pública, pero sólo el usuario dueño del par de claves conoce la privada. Por ejemplo, el acceso por SSH se puede realizar mediante el intercambio de este tipo de claves. La clave pública se encontraría en el archivo *authorized_keys* y el usuario proveería la clave privada en el momento de la conexión.

Concern: son los intereses que tienen los *stakeholders* de un proyecto. En el ámbito de este trabajo, son aquellas inquietudes que cada *stakeholder* manifiesta acerca de la arquitectura.

Contenedores (Containers): en el caso de las máquinas virtuales, el virtualizador se encarga de virtualizar distintos elementos de *hardware*, como CPU, memoria, disco, buses, etc. Mientras que en el caso de los contenedores, el sistema operativo *virtualiza* elementos del sistema operativo, como la estructura de directorios, los usuarios del sistema o el árbol de procesos. La tecnología de contenedores es nativa de Linux. Ver *Jails*.

Crowdsourcing: es la práctica mediante la cual se obtiene algún tipo de servicio solicitando la contribución de un gran número de personas, generalmente a través de Internet. Divide el trabajo entre un gran número de participantes para obtener un resultado acumulativo.

Default gateway: en una red TCP/IP, es un nodo al que los demás hosts de esa red le envían los paquetes cuando no tienen una ruta que coincida con la IP destino del paquete.

Deployment (de una aplicación): son todas las actividades o tareas que se llevan a cabo para que una aplicación se pueda utilizar. Entre las actividades típicas se encuentran: configurar el servidor web, crear la base de datos o actualizar una existente, hacer un backup de la configuración anterior, entre otras.

Docker: es una aplicación utilizada para el manejo de contenedores.

File system: indica cómo se almacenan y recuperan los datos en un sistema informático.

Firewall: es un sistema de seguridad que monitorea y controla todo el tráfico entrante y saliente de una red. Se lo utiliza para establecer un perímetro de seguridad entre una red interna confiable y una red externa no confiable.

Fitness function: provee una manera objetiva de evaluar la integridad de alguna característica del software. Por ejemplo, dado el requerimiento "todas las consultas a la API deben tardar menos de 10ms", se puede implementar un test que mida el tiempo de respuesta y falle si es mayor a 10ms. El test, en este ejemplo, es la *fitness function*. Otro ejemplo: dado el requerimiento "todos los métodos públicos deben estar documentados", se puede correr un test que verifique luego de una modificación en el código si hay métodos públicos sin documentar.

High availability (HA) o Alta disponibilidad: la disponibilidad de un sistema describe el periodo de tiempo durante el cual brinda servicio. HA es una cualidad del sistema que aspira a asegurar un determinado nivel operacional (generalmente *uptime*) por un período mayor al negociado. Para lograrlo se recurren a diversas técnicas, como por ejemplo tener réplicas de servidores.

Jails: es una implementación particular de FreeBSD de lo que se llama "virtualización a nivel de sistema operativo", en donde el sistema operativo virtualiza componentes como el árbol de procesos o los usuarios del sistema, permitiendo a los administradores disponer de varios mini-sistemas independientes llamados *jails*. La primera versión de esta tecnología apareció en FreeBSD 4, en el año 2000. Ver *Contenedores*.

Logging: los sistemas operativos y las aplicaciones almacenan diferentes eventos en archivos llamados *logs*. El proceso por el cual se mantienen esos archivos se llama *logging* e involucra tareas como rotar los archivos y comprimirlos para que no llenen el disco del servidor, copiar esos archivos a otros servidores para ser analizados, estandarizar el formato de los archivos, porque no todas las aplicaciones escriben los logs de la misma manera, crear gráficos y alertas en base a esos logs, entre otras tareas.

Marcacion en GitHub (starring): es una funcionalidad de Github que permite a los usuarios marcar un repositorio que les resulta interesante, similar a los "Favoritos" del navegador.

Montar un *file system*: según la página del manual de mount(8), "Todos los ficheros accesibles en un sistema Unix están dispuestos en un gran árbol, la jerarquía de ficheros, con la raíz en /. Estos ficheros pueden estar distribuidos sobre varios dispositivos. La orden mount sirve para pegar el sistema de ficheros encontrado en algún dispositivo al gran árbol de ficheros"[4].

NAT: es una técnica para mapear un espacio de direcciones IP en otro. El método consiste en modificar la información de direccionamiento en el encabezado IP de los paquetes mientras atraviesan un dispositivo de ruteo.

Network File System (NFS): es un protocolo que sirve para acceder, usando la red, un *file system* remoto de forma similar a como se accede el almacenamiento local. Desde el punto de vista del usuario, casi no hay diferencia entre usar un *file system* local y uno remoto montado por NFS.

Prompt: es una secuencia de caracteres usados en la linea de comandos para indicar que el sistema está listo para aceptar comandos. Por ejemplo, un *prompt* puede tener la forma genérica **username@hostname:directorio$**. El usuario **matias** situado en el directorio **/etc/bacula** del host **bacula** verá un *prompt* como este: **matias@bacula:/etc/bacula$**

Pull (modelo): en el contexto de las herramientas de automatización, bajo el modelo pull los servidores buscan la configuración en un servidor central. Ejemplos de tales aplicaciones son Chef y Puppet.

Push (modelo): en el contexto de las herramientas de automatización, bajo el modelo push los administradores "empujan" la configuración hacia los servidores. No existe un servidor central. Ejemplos de estas aplicaciones son Ansible y Salt.

Round robin: es un algoritmo de planificación simple y fácil de implementar. En el caso de los sistemas operativos que lo utilizan para asignar recursos de CPU, es libre de inanición. En el contexto de este trabajo, las peticiones que recibe el proxy reverso se distribuyen entre los servidores web cíclicamente.

Secure Sockets Layer (SSL): Ver TLS.

Shell: es un intérprete de comandos textual. Los usuarios le indican a la computadora las operaciones que debe realizar mediante la escritura de comandos textuales. Existen varias *shells*, como **ash**, **bash**, **zsh**, entre otras.

Single Sign On (SSO): algunos sistemas, aunque independientes, están relacionados entre sí. Si estos sistemas implementan algún mecanismo de SSO, un usuario necesita un sólo nombre de usuario y contraseña para acceder a todos estos sistemas. De hecho, al ingresar a uno de estos sistemas queda autorizado para acceder a los demás sin necesidad de volver a ingresar sus credenciales.

[4] https://man.cx/mount(8)/es

SSH: es un protocolo utilizado para acceder a dispositivos remotamente. Provee un medio cifrado sobre una red no confiable, como es Internet.

Stakeholders: es un individuo, grupo u organización, que puede afectar o ser afectado por una decisión, actividad o resultado de un proyecto. Para una definición más detallada se recomienda el correspondiente artículo de Wikipedia[5], donde se cita la definición del Project Management Institute (PMI) y de la ISO 21500.

Trade-off: se refiere a una solución de compromiso en donde se degrada o pierde una cualidad en favor de otra. Por ejemplo, se degrada el desempeño para ganar seguridad mediante el cifrado de los datos.

Transport Layer Security (TLS): es un protocolo de criptografía que provee una capa de cifrado sobre una red TCP/IP. En la práctica, datos que de otra forma viajarían en texto plano con el uso de TLS lo hacen sobre un medio cifrado. TLS es el reemplazo de SSL, que se lo considera *deprecated* por la IETF.

Uninterruptible power supply (UPS): es un dispositivo eléctrico que provee energía cuando la alimentación principal falla. Existen de diferentes capacidades y el tiempo durante el cual proveerá energía depende de la carga que tenga (la cantidad de dispositivos que estén conectados).

Uptime: es una métrica que indica el tiempo que un servidor o softwware ha estado disponible y brindando servicio.

Usuario root: es el usuario que tiene todos los permisos en todos los modos de ejecución en los sistemas UNIX y derivados. Es decir que puede acceder a cualquier archivo y ejecutar cualquier aplicación en el sistema.

Utility tree: traducen las necesidades del negocio en escenarios concretos que tratan con los atributos de calidad. El proceso de crear y refinar los *utility trees* guía a los principales *stakeholders* en considerar y priorizar todas las fuerzas, actuales y futuras, que modelan la arquitectura.

Variable $HOME: es una variable de ambiente cuyo valor es el directorio *home* del usuario. En los sistemas multiusuario, el directorio *home* contiene los archivos de un usuario dado. Por ejemplo, para el usuario **matias** el valor de la variable *$HOME* es **/home/matias**.

[5] https://en.wikipedia.org/wiki/Project_stakeholder