UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MÁRCIO BARBOSA DE CARVALHO

Um *Framework* para a Construção Automatizada de *Cloud Monitoring Slices* Baseados em Múltiplas Soluções de Monitoramento

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Lisandro Zambenedetti Granville
Orientador

Porto Alegre, março de 2015

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Carvalho, Márcio Barbosa de

Um *Framework* para a Construção Automatizada de *Cloud Monitoring Slices* Baseados em Múltiplas Soluções de Monitoramento / Márcio Barbosa de Carvalho. – Porto Alegre: PPGC da UFRGS, 2015.

87 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2015. Orientador: Lisandro Zambenedetti Granville.

1. Computação em nuvem. 2. Monitoramento. 3. Soluções de monitoramento. 4. Framework. I. Granville, Lisandro Zambenedetti. II. Título.

“No fim tudo dá certo,
e se não deu certo é
porque ainda não
chegou ao fim”
— FERNANDO SABINO, ESCRITOR

AGRADECIMENTOS

Gostaria de agradecer a toda minha família por todo apoio e incentivo. Em especial, aos meus pais (Pedro *"in memoriam"* e Vera) e a minha avó (Eva) pela educação e valores ensinados, sem os quais eu seria ninguém. Devo o pouco que eu já alcancei a vocês. À minha irmã Raquel pela parceria nos bons e maus momentos. À minha esposa Camila pelo companheirismo, paciência, dedicação e força para superar os momentos difíceis. **Te amo, *minina*!**

Agradeço ao professor Lisandro por me aceitar como orientando para o Trabalho de Graduação. Agradeço pelo incentivo (e dedicação) na escrita do artigo sobre o TG submetido para o IM 2011. Sem a aceitação deste artigo, eu não teria ingressado no mestrado. Também agradeço pela confiança depositada, pelo incentivo contínuo e pelo grande aprendizado proporcionado nestes anos de orientação. Tenho muito orgulho de ser teu orientando pela tua excelência acadêmica, mas também pela sabedoria e sensatez com que orienta os teus alunos.

Agradeço aos professores Luciano Paschoal Gaspary e Marinho P. Barcellos pela qualidade das disciplinas ministradas no PPGC e no INF. O rigor cobrado e a qualidade das discussões certamente me ajudaram a evoluir como estudante e profissional. Agradeço também à professora Liane Tarouco pelo apoio na apresentação de artigo durante o ISCC 2014 (é bom ver rostos conhecidos na plateia).

Agradeço aos amigos do Grupo de Redes pela parceria (leia-se também bebedeira) no IM 2011 em Dublin (Roben Lunardi, Rafael Esteves, Pedro Jedi, Bruno Dalmazo e Rodrigo Mansilha). Ao Wanderson Paim pela grande parceria nas apresentações de trabalhos nas disciplinas do PPGC. Àqueles que tive oportunidade de trabalhar (e principalmente aprender) na escrita de artigos (Rafael Esteves, Guilherme Rodrigues, Juliano Wickboldt e Clarissa Marquezan). Um agradecimento especial para Rafael Esteves e Guilherme Rodrigues pelas revisões, críticas e sugestões que contribuíram (e muito) para o aprimoramento desta dissertação.

RESUMO

Computação em nuvem é um paradigma em que provedores oferecem recursos computacionais como serviços, que são contratados sob demanda e são acessados através da Internet. Os conjuntos de recursos computacionais contratados são chamados de *cloud slices*, cujo monitoramento fornece métricas essenciais para atividades como a operação da infraestrutura, verificação do cumprimento de SLAs e medição da qualidade do serviço percebida pelos usuários. Além disso, o monitoramento também é oferecido como serviço para os usuários, que podem contratar métricas ou serviços de monitoramento diferenciados para seus *cloud slices*. O conjunto de métricas associadas a um *cloud slice* juntamente com as configurações necessárias para coletá-las pelas soluções de monitoramento é chamado de *monitoring slice*, cuja função é acompanhar o funcionamento do *cloud slice*. Entretanto, a escolha de soluções para serem utilizadas nos *monitoring slices* é prejudicada pela falta de integração entre soluções e plataformas de computação em nuvem. Para contornar esta falta de integração, os administradores precisam implementar *scripts* geralmente complexos para coletar informações sobre os *cloud slices* hospedados na plataforma, descobrir as operações realizadas na plataforma, determinar quais destas operações precisam ser refletidas no monitoramento de acordo com as necessidades do administrador e gerar as configurações dos *monitoring slices*.

Nesta dissertação é proposto um *framework* que mantém *monitoring slices* atualizados automaticamente quando *cloud slices* são criados, modificados ou destruídos na plataforma de nuvem. Neste *framework*, os *monitoring slices* são mantidos de acordo com regras predefinidas pelos administradores oferecendo a flexibilidade que não está disponível nas soluções de monitoramento atuais. Desta forma, o desenvolvimento de *scripts* complexos é substituído pela configuração dos componentes do *framework* de acordo com as necessidades dos administradores em relação ao monitoramento. Estes componentes realizam a integração do *framework* com as plataformas e soluções de monitoramento e podem já ter sido desenvolvidos por terceiros. Caso o componente necessário não esteja disponível, o administrador pode desenvolvê-lo facilmente aproveitando as funcionalidades oferecidas pelo *framework*. Para avaliar o *framework* no contexto de nuvens do modelo IaaS, foi desenvolvido o protótipo chamado FlexACMS (*Flexible Automated Cloud Monitoring Slices*). A avaliação do FlexACMS mostrou que o tempo de resposta para a criação de *monitoring slices* é independente do número de *cloud slices* no *framework*. Desta forma, foi demonstrada a viabilidade e escalabilidade do FlexACMS para a criação de *monitoring slices* para nuvens IaaS.

Palavras-chave: Computação em nuvem, monitoramento, soluções de monitoramento, framework.

A Framework for Automatic Building of Cloud Monitoring Slices based on Multiple Monitoring Solutions

ABSTRACT

Cloud computing is a paradigm that providers offer computing resources as services, which are acquired on demand and are accessed through the Internet. The set of acquired computing resources are called cloud slices, whose monitoring offers essential metrics for activities as infrastructure operation, SLA supervision, and quality of service measurement. Beyond, the monitoring is also offered as a service to users, that can acquire both differentiated metrics or monitoring services to their cloud slices. The set of metrics associated to a cloud slice and the required configuration to collect them by monitoring solutions is called monitoring slice, whose function is keep up with the cloud slice functioning. However, the monitoring solution choice to compose monitoring slices is harmed by lack of integration between solutions and cloud platforms. To overcome this lack of integration, the administrators need to develop scripts usually complex to collect information about cloud slices hosted by the platform, to discover the operations performed in the platform, to determine which operations need to be reflected in the monitoring according to the administrator's needs, and to generate the monitoring slice configuration.

This dissertation proposes a framework that keeps monitoring slices updated automatically when cloud slices are created, modified, or destroyed in the cloud platform. In this framework, the monitoring slices are kept according to rules defined by administrators, which offers the flexibility that is not available in current monitoring solutions. In this way, the framework replaces the development of complex scripts by the configuration of framework's components according to administrator's needs in regards to monitoring. These components perform the framework integration with platforms and monitoring solutions and may be already developed by third parties. If required component is not available, the administrator can easily develop it availing functionalities offered by the framework. In order to evaluate the framework in the context of IaaS clouds, a prototype called FlexACMS (Flexible Automated Cloud Monitoring Slices) was developed. The FlexACMS evaluation showed that response time to create monitoring slices is independent of the number of cloud slices in the framework. In this way, the FlexACMS feasibility and scalability was demonstrated for creation of monitoring slices for IaaS clouds.

Keywords: Cloud computing, monitoring, monitoring solutions, framework.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AWS	Amazon Web Services
CIMI	Cloud Infrastructure Management Interface
CoC	Convention over Configuration
CPU	Central Processing Unit
DDS	Data Distribution Service
DMTF	Distributed Management Task Force
DNS	Domain Name System
EBS	Elastic Block Store
E/S	Entrada/Saída
EC2	Elastic Cloud Computing
ELB	Elastic Load Balancing
FlexACMS	Flexible Automated Cloud Monitoring Slices
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
KVM	Kernel-based Virtual Machine
IaaS	Infrastructure as a Service
ICMP	Internet Control Message Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
MaaS	Monitoring as a Service
MRTG	Multi Router Traffic Grapher
MVC	Model-View-Controller
MVM	Monitoring Virtual Machine
NRPE	Nagios Remote Plugin Executor
OCCI	Open Cloud Computing Interface

OGF	Open Grid Forum
OpenStack API	OpenStack Application Programming Interface
OVF	Open Virtualization Format
PaaS	Platform as a Service
PHP	PHP: Hypertext Preprocessor
QoS	Quality of Service
REST	REpresentational State Transfer
RDS	Relational Database Service
S3	Simple Storage Service
SaaS	Software as a Service
SDN	Software-defined Networking
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
TCP ACK	Transmission Control Protocol Acknowledgement
UML	User Mode Linux
VXDL	Virtual Resources and Interconnection Networks Description Language
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Computação em nuvem é um paradigma em que provedores disponibilizam recursos computacionais como serviços que são contratados sob demanda e acessados através da Internet (VAQUERO et al., 2008) (ZHANG; CHENG; BOUTABA, 2010). Recentemente, este paradigma tornou-se popular em virtude de oferecer recursos acessíveis em diversos níveis de abstração, de acordo com o modelo de negócio da nuvem. No modelo de negócio *Infrastructure as a Service* (IaaS), por exemplo, é possível contratar recursos como processamento, armazenamento e memória. No modelo *Platform as a Service* (PaaS) é possível contratar recursos como servidores de aplicação e plataformas de desenvolvimento. Já no modelo *Software as a Service* (SaaS) é possível contratar recursos de alto nível como aplicativos. Estes recursos são cobrados de acordo com a sua utilização e podem ser ampliados ou reduzidos ao longo do tempo, de acordo com a demanda do usuário. Esta última característica é chamada de elasticidade e permite aos usuários ajustar o volume dos recursos computacionais contratados de acordo com sua demanda.

Além de disponibilizar os recursos computacionais para os usuários, o provedor da nuvem também pode ser responsável pelo gerenciamento destes recursos e da infraestrutura que os suporta (MELL; GRANCE, 2011). O provedor da nuvem deve garantir, por exemplo, que a infraestrutura seja tolerante a falhas e assegurar que parâmetros estabelecidos nos acordos de nível de serviço (SLAs - *Service Level Agreements*) sejam cumpridos. Os SLAs possuem cláusulas que penalizam economicamente o provedor caso os acordos não sejam cumpridos, de maneira que os usuários recebem descontos na mensalidade dos recursos afetados ocasionando perdas consideráveis ao provedor (BUYYA et al., 2009). Além disso, o não cumprimento de SLAs afeta negativamente a reputação do provedor, o que pode prejudicar negócios futuros. Logo, o gerenciamento adequado do ambiente é uma atividade crítica para a operação da infraestrutura e para o negócio dos provedores.

Neste contexto, dentre as atividades de gerenciamento, o monitoramento é importante para os provedores de nuvem em diversos aspectos. Primeiramente, é fundamental para que falhas sejam detectadas rapidamente para que sejam resolvidas antes que SLAs sejam violados. Também é fundamental para acompanhar a utilização da infraestrutura para adequá-la à demanda dos usuários, o que evita o desperdício ou a sobrecarga de recursos físicos (YAZIR et al., 2010). Já o monitoramento dos recursos computacionais disponibilizados aos usuários oferece uma visão da qualidade do serviço (QoS - *Quality of Service*) percebida pelos usuários (BONIFACE et al., 2010). Além disso, o monitoramento dos recursos disponibilizados aos usuários tornou-se parte do negócio dos provedores, que podem oferecer métricas e serviços de monitoramento diferenciados através do modelo de negócio *Monitoring as a Service* (MaaS) (MENG; LIU, 2013). Logo, o monitoramento impacta diretamente no lucro dos provedores, pois auxilia que SLAs sejam cumpridos e possibilita ganhos extras através de MaaS.

As soluções de monitoramento para ambientes computacionais em nuvem devem atender a requisitos relacionados à infraestrutura do provedor e aos recursos oferecidos como serviços aos usuários. Neste sentido, Aceto *et al.* (2013) conduziram um estudo sobre os requisitos necessários e quais soluções de monitoramento os satisfazem. Através deste estudo, pode-se concluir que não existe uma única solução de monitoramento para computação em nuvem que satisfaça todos os requisitos necessários. Além disso, os administradores necessitam de funcionalidades do sistema de monitoramento para auxiliar a operação do ambiente, tais como gráficos, relatórios e notificações através de diferentes meios de comunicação. Logo, os requisitos e a presença de algumas funcionalidades afetam a escolha das soluções de monitoramento que serão utilizadas.

Surgem duas abordagens para que soluções de monitoramento atendam aos requisitos e ofereçam as funcionalidades necessárias para ambientes de computação em nuvem. A primeira é utilizar uma única solução de monitoramento desenvolvida ou adaptada de outra existente de modo que atenda todas as necessidades e funcionalidades. A segunda é possibilitar que várias soluções de monitoramento sejam utilizadas para agregar requisitos e funcionalidades. Entretanto, pode-se observar em ambientes tradicionais que os administradores costumam utilizar diversas soluções de monitoramento para explorar funcionalidades específicas como gráficos e notificações, mesmo que existam soluções de monitoramento completas à disposição para ambientes tradicionais. Logo, pode-se afirmar que mesmo com soluções completas para ambientes computacionais em nuvem, os administradores continuarão utilizando diversas soluções de monitoramento para atender determinadas necessidades. Portanto, a segunda abordagem não pode ser ignorada para a construção de soluções de monitoramento para ambientes de computação em nuvem.

No modelo de nuvens IaaS é necessário monitorar a infraestrutura do ambiente e os conjuntos de recursos contratados pelos usuários, tais como processamento, armazenamento e memória. Este conjunto de recursos é chamado de *slice* ou *cloud slice* (WICKBOLDT et al., 2014). O monitoramento da infraestrutura do ambiente é semelhante ao realizado em ambientes tradicionais com suporte à virtualização, em que são monitoradas as plataformas de virtualização (neste caso, de nuvem) e os recursos físicos do ambiente (CARVALHO; GRANVILLE, 2011). Por outro lado, o monitoramento dos *cloud slices* impõe novos desafios devido à sua dinamicidade, pois *cloud slices* podem ser criados ou destruídos a qualquer momento e o monitoramento deve adaptar-se a estas operações. Por exemplo, após a criação de um *cloud slice* é necessário configurar soluções de monitoramento para coletar o conjunto de métricas associadas aos seus recursos. Este conjunto de métricas e as configurações necessárias para coletá-las é chamado de *monitoring slice*, cuja função é acompanhar o funcionamento do *cloud slice*[1] (CARVALHO et al., 2013).

Existem soluções de monitoramento que são integradas às plataformas de nuvem e são configuradas automaticamente quando *cloud slices* são criados, modificados ou destruídos (VAN VLIET; PAGANELLI, 2011). Por outro lado, existem outras soluções que não são integradas, o que dificulta a configuração do monitoramento para acompanhar as operações ocorridas sobre os *cloud slices* nas plataformas. Entre as soluções que não são integradas às plataformas, existem aquelas que utilizam um processo de descoberta para, por exemplo, incluir novos dispositivos conectados à rede em suas configurações. Entretanto, esta descoberta deveria ser executada frequentemente para acompanhar as operações sobre *cloud slices* nas plataformas, o que seria demasiadamente custoso pois trata-se de um processo que consome muitos recursos da rede. Visando contornar res-

[1]Preferiu-se utilizar os termos em inglês *"cloud slice"* e *"monitoring slice"* porque as traduções livres "fatia de nuvem"e "fatia de monitoramento" não são utilizadas pela comunidade.

trições de integração, os administradores costumam desenvolver *scripts* para automatizar tarefas. Entretanto, o desenvolvimento destes *scripts* é complexo, pois deve-se descobrir quais *cloud slices* sofreram alterações desde a última execução e coletar informações da plataforma de nuvem sobre estes *cloud slices*. Posteriormente, os *scripts* podem utilizar estas informações para atualizar as configurações das soluções de monitoramento de acordo com as operações efetuadas sobre os *cloud slices*.

Dentre os objetivos desta dissertação está implementar uma solução que: *(i)* permita que administradores de ambientes de computação em nuvem utilizem as soluções de monitoramento que atendam as suas necessidades independentemente da integração existente entre solução de monitoramento e plataforma de computação em nuvem; e *(ii)* automatize tarefas de configuração das soluções de monitoramento de acordo com as operações realizadas na plataforma de computação em nuvem respeitando regras predefinidas pelos administradores do ambiente de computação em nuvem.

Nesta dissertação é proposta uma arquitetura para um *framework* que mantém *monitoring slices* atualizados automaticamente quando *cloud slices* são criados, modificados ou destruídos na plataforma de nuvem. Este *framework* suporta diversas plataformas de nuvem e soluções de monitoramento independentemente da integração existente entre elas. O *framework* é responsável pela coleta e organização de informações das plataformas de nuvem, pela detecção de operações que ocorreram na plataforma e pela configuração das soluções de monitoramento de acordo com regras definidas pelo administrador.

Para avaliar a arquitetura proposta em uma nuvem do modelo IaaS, foi desenvolvido o protótipo FlexACMS (*Flexible Automated Cloud Monitoring Slices*) (CARVALHO et al., 2013) em duas versões que consideram apenas a criação de *monitoring slices*, embora outras ações sejam conceitualmente suportadas pelo *framework*. A primeira versão utiliza uma abordagem serial com desempenho semelhante ao que administradores conseguiriam desenvolvendo *scripts* de automação próprios. Na segunda versão (CARVALHO et al., 2014), esta arquitetura foi estendida com o intuito de melhorar o desempenho explorando o fato de que *monitoring slices* são independentes, *i.e.*, a criação de um *monitoring slice* não afeta a criação de outro *monitoring slice*. Desta forma, *monitoring slices* podem ser configurados de maneira paralela. Além disso, a extensão explora o paralelismo disponível nas atuais arquiteturas de computadores. Ambas versões do FlexACMS foram avaliadas e comparadas através de experimentos em ambientes reais que utilizam a plataforma de nuvem OpenStack (FIFIELD et al., 2014) e as ferramentas de monitoramento Nagios (KOCJAN, 2014) e *Multi Router Traffic Grapher* (MRTG) (SHIPWAY; OETIKER, 2010) que não são nativamente integradas ao OpenStack. Além disso, a arquitetura estendida foi submetida a um experimento para comprovar a escalabilidade da solução em relação ao seu tempo de resposta em cenários com grande número de *cloud slices*.

O restante desta dissertação é organizado como segue. No capítulo 2 são apresentados conceitos sobre computação em nuvem, bem como são apresentadas soluções do estado-da-arte para o monitoramento de ambientes computacionais em nuvem. No capítulo 3 são apresentados os detalhes da solução proposta, como modelo de informação e as arquiteturas das duas versões do FlexACMS. No capítulo 4 são apresentados os detalhes de implementação dos componentes da arquitetura do FlexACMS, bem como um apanhado das tecnologias envolvidas no funcionamento da solução. No capítulo 5 são apresentados os resultados das avaliações em relação ao tempo de resposta, consumo de processamento, memória e banda de comunicação e os resultados da avaliação de escalabilidade da arquitetura estendida. No capítulo 6 são apresentadas as conclusões finais e trabalhos futuros no contexto deste trabalho.

2 TRABALHOS RELACIONADOS

Neste capítulo são apresentados conceitos e soluções relacionados a esta dissertação. Na Seção 2.1 são apresentados conceitos básicos de computação em nuvem, como nuvens públicas e privadas. Na Seção 2.2 são apresentadas as principais funcionalidades de plataformas de computação em nuvem, bem como exemplos de plataformas e de serviços de provedores de nuvem. Na Seção 2.3 são apresentados conceitos relacionados ao monitoramento de ambientes de computação em nuvem, bem como são apresentados os requisitos que as soluções de monitoramento devem possuir. Na Seção 2.4 são apresentados exemplos destas soluções e um panorama dos requisitos atendidos pelas soluções de monitoramento atuais. Na Seção 2.5 são discutidas as possíveis abordagens para ampliar os requisitos atendidos pelas soluções de monitoramento, bem como é discutida a necessidade de configuração automática destas soluções para computação em nuvem.

2.1 Computação em nuvem

Nesta seção são apresentados conceitos básicos sobre computação em nuvem. São apresentadas definições, cenários típicos, características oferecidas pelos ambientes de computação em nuvem e os diferentes tipos de nuvens.

A esperança de que se tornaria realidade o antigo sonho de ter serviços de computação contratados como utilidade (como o telefone e a eletricidade) elevou rapidamente a popularidade da computação em nuvem (ARMBRUST et al., 2010). Entretanto, houve muita confusão ao caracterizar computação em nuvem, o que levou a um grande número de definições, como por exemplo: a definição do NIST (MELL; GRANCE, 2011), a definição de Vaquero *et al.* (2008) (baseada em 20 definições anteriores), e a definição de Zhang, Cheng e Boutaba (2010). Além de diversas definições, existem similaridades entre computação em nuvem e outros paradigmas como computação em grade (FOSTER et al., 2008), o que torna mais difícil determinar claramente o que é computação em nuvem. Esta dificuldade está principalmente relacionada ao fato de não existirem novas tecnologias envolvidas (ZHANG; CHENG; BOUTABA, 2010), porém tratar-se de uma nova abordagem para oferecer recursos computacionais que utiliza as tecnologias existentes. Abaixo é apresentada uma definição de computação em nuvem que apresenta as principais características presentes nas definições anteriores.

Computação em nuvem é um paradigma onde recursos computacionais virtualizados (*e.g.* hardware, plataformas, software) são oferecidos como serviço por provedores e acessados pelos usuários através da Internet. Estes recursos computacionais podem ser ajustados dinamicamente ao longo do tempo para adequar-se à demanda do usuário e são cobrados de acordo com sua utilização. Por outro lado, o provedor oferece garantias em relação a estes recursos que são estabelecidas através de SLAs (VAQUERO et al., 2008).

Os recursos contratados também podem ser rapidamente aprovisionados e liberados com mínimo esforço por parte do usuário ou interação com administradores e operadores do provedor da nuvem (ZHANG; CHENG; BOUTABA, 2010). A aparente falta de limite na capacidade de alocação oferecida pela nuvem proporciona a ilusão de que os recursos contratados podem ser infinitos (ARMBRUST et al., 2010).

Um cenário típico para um ambiente de computação em nuvem pública é ilustrado na Figura 2.1. Neste cenário, o provedor de infraestrutura é responsável por fornecer e gerenciar os recursos físicos e a infraestrutura que os suporta. O provedor de infraestrutura utiliza uma plataforma de gerenciamento de nuvem para fornecer os recursos contratados pelos provedores de serviço. Esta plataforma fornece interfaces de programação que permitem aos provedores de serviço gerenciar os recursos virtuais contratados e disponibilizar os serviços para os usuários finais. Por se tratar de uma nuvem pública, os recursos contratados e serviços oferecidos são acessados pelos provedores de serviço e usuários finais através da Internet, respectivamente.

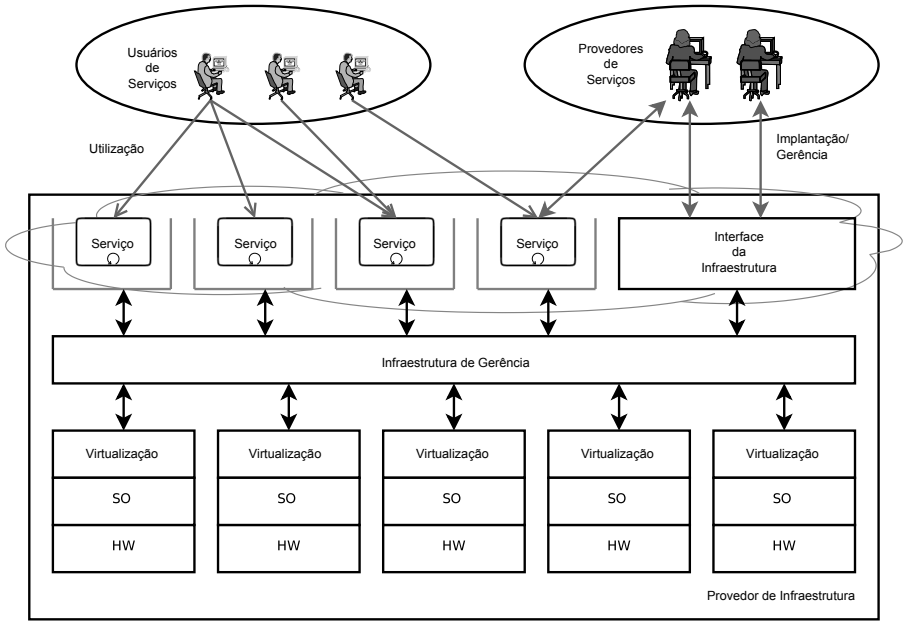


Figura 2.1: Cenário típico de um ambiente de computação em nuvem (traduzido de Vaquero *et al.* (2008))

O paradigma de computação em nuvem apresenta características como virtualização, elasticidade, modelo flexível de cobrança, e SLAs, conforme apresentado nas definições anteriores. Entretanto, a presença destas características não é suficiente ou necessária para caracterizar uma solução como de computação em nuvem (RHOTON, 2009). A seguir são apresentadas características que são normalmente encontradas em ambientes de nuvens públicas, cuja presença pode ser utilizada para definir se uma solução adere ao paradigma de computação em nuvem (RHOTON, 2009).

Serviços fora da infraestrutura do usuário - Característica relacionada ao termo nuvem, que neste caso representa a Internet. Em computação em nuvem, os recursos são hospedados fora da infraestrutura do usuário e são acessados pela Internet. Outro aspecto relacionado é que na maioria das vezes os usuários não conhecem a localidade, região ou país onde os recursos contratados estão hospedados. Este aspecto leva a preocupações de segurança e, nesse sentido, podem ser estabelecidas restrições quanto ao local ou país onde os dados dos usuários podem estar armazenados (SUBASHINI; KAVITHA, 2011).

Elasticidade - Característica relacionada à possibilidade de aquisição (ou liberação) de recursos computacionais para suportar aumento (ou diminuição) da demanda do usuário de maneira rápida. Desta forma, oferece grande vantagem em relação ao modelo tradicional em que os recursos computacionais são de propriedade dos usuários, pois permite aumento da capacidade computacional de maneira rápida e sem preocupações relacionadas à aquisição e instalação de hardware e software (WANG et al., 2010).

Tarifação flexível - Computação em nuvem utiliza um modelo de tarifação flexível baseado na utilização real ou na quantidade de recursos computacionais reservados pelo usuário. Este modelo pode basear-se em dados coletados pelo monitoramento para uma tarifação precisa em relação aos recursos que foram efetivamente utilizados (ELMROTH et al., 2009); ou pode basear-se na quantidade de recursos reservados pelo usuário independentemente da sua utilização (BUYYA et al., 2009). A tarifação flexível em conjunto com a elasticidade permite que os usuários ajustem o custo dos recursos computacionais contratados à sua demanda (ARMBRUST et al., 2009).

Virtualização - As plataformas de gerenciamento de nuvem costumam utilizar virtualização como abstração para oferecer recursos computacionais aos usuários. A virtualização flexibiliza a alocação dos recursos contratados pelos usuários na infraestrutura do provedor, pois fornece isolamento e permite que recursos virtuais compartilhem o mesmo recurso físico. Desta forma, o provedor utiliza eficientemente os recursos da infraestrutura, pois reduz o desperdício diminuindo a quantidade de recursos ociosos (ZHANG; CHENG; BOUTABA, 2010). Além disso, a virtualização oferece capacidade de tolerância a falhas, pois os recursos virtualizados são independentes do hardware subjacente e podem ser migrados para contornar eventuais falhas e sobrecargas (LU; CHIUEH, 2009).

Serviços - Os recursos computacionais são entregues pelos provedores aos usuários como serviço. O modelo de nuvem é definido pelo nível de abstração dos serviços oferecidos aos usuários. Por exemplo, no modelo de nuvem IaaS (*Infrastructure as a Service*) recursos computacionais de infraestrutura como capacidade de processamento, memória e armazenamento são oferecidos como serviço. No modelo IaaS, os recursos de infraestrutura são normalmente entregues na forma de conjuntos de máquinas virtuais que também são chamados de *cloud slices* (WICKBOLDT et al., 2014).

Gerenciamento simplificado - O gerenciamento realizado pelos usuários é simplificado porque fica restrito ao nível de acesso aos recursos que o usuário possui. Por exemplo, o gerenciamento pode ser restrito às operações permitidas pela interface disponibilizada pelo provedor; ou pode ser restrito ao gerenciamento dos serviços que tenha implementado sobre os recursos contratados (ARMBRUST et al., 2009). Por outro lado, o gerenciamento da plataforma e da infraestrutura que suportam os recursos é de responsabilidade dos provedores, que devem mantê-los em funcionamento de acordo com os parâmetros estabelecidos nos SLAs (BASET, 2012).

Recursos economicamente acessíveis - Os recursos computacionais tornam-se economicamente acessíveis para usuários que não poderiam arcar com o alto investimento inicial para construir uma infraestrutura de tecnologia de informação própria (ZHANG; CHENG; BOUTABA, 2010). Além do custo inicial, estas infraestruturas são complexas e exigem equipes capacitadas para seu gerenciamento. Além disso, o usuário não está exposto aos riscos envolvidos na posse da infraestrutura (ARMBRUST et al., 2010), como a deterioração natural e a substituição de equipamentos defeituosos.

Compartilhamento de recursos - A infraestrutura do provedor é compartilhada por múltiplos usuários, o que exige preocupações relacionadas à segurança, como o isolamento entre estes recursos. Por outro lado, o compartilhamento de recursos permite a uti-

lização eficiente dos recursos da infraestrutura do provedor para maximizar seus lucros. A eficiência pode estar relacionada à utilização plena da infraestrutura, o que diminui a quantidade de equipamentos reduzindo o custo total da infraestrutura; ou em relação à eficiência energética da infraestrutura para evitar que equipamentos consumam mais energia por estarem sobrecarregados (GOUDARZI; GHASEMAZAR; PEDRAM, 2012).

Gerenciamento no âmbito dos serviços - Os recursos computacionais contratados são oferecidos como serviço, que possuem qualidade garantida pelos provedores aos usuários através de SLAs. Estas garantias são expressas com base em parâmetros no âmbito dos serviços, ou seja, no mesmo nível dos recursos contratados (BASET, 2012). Por exemplo, no caso de uma nuvem IaaS, o SLA pode definir que as máquinas virtuais contratadas devem possuir disponibilidade de 99.9% ao mês. Isto facilita o relacionamento entre usuário e provedor, pois os SLAs abstraem toda a complexidade da infraestrutura do provedor que não possui relevância para o usuário. No exemplo, o SLA não especifica que determinados componentes da infraestrutura do provedor, como servidores de armazenamento, virtualização ou dispositivos de rede possuam a disponibilidade acordada, mas fica implícito para o provedor que a sua infraestrutura deverá utilizar mecanismos de tolerância a falhas que garantam tal disponibilidade.

Apesar das definições e características apresentadas anteriormente serem baseadas principalmente em conceitos de nuvens públicas, o paradigma de computação em nuvem apresenta características que também são interessantes para infraestruturas privadas. Organizações podem criar sua própria infraestrutura de nuvem privada para explorar estas características, tais como elasticidade e virtualização. Além disso, as plataformas de nuvens privadas podem ser integradas às nuvens públicas o que permite a combinação de recursos computacionais privados e públicos. Desta forma, uma organização pode operar com recursos próprios em períodos de demanda normal e alcançar escalabilidade com recursos da nuvem pública em períodos de aumento da demanda (SOTOMAYOR et al., 2009). Esta abordagem é denominada nuvem híbrida e explora tanto os benefícios de infraestruturas próprias como os benefícios das nuvens públicas.

2.2 Plataformas de administração e provedores de nuvem

Nesta seção são apresentados conceitos relacionados às plataformas de administração e aos provedores de computação em nuvem. São apresentadas as funcionalidades típicas fornecidas pelas plataformas, bem como exemplos de plataformas e de serviços oferecidos por provedores de computação em nuvem.

Existem plataformas de gerenciamento que permitem a provedores e organizações explorarem os benefícios do paradigma de computação em nuvem. As plataformas oferecem funcionalidades que auxiliam o gerenciamento, oferecimento e utilização de recursos computacionais. Para ilustrar estas funcionalidades e componentes normalmente encontrados nas plataformas, a Figura 2.2 foi desenvolvida para apresentar uma arquitetura genérica de plataformas de administração de nuvem. Os componentes desta arquitetura e suas funcionalidades são detalhados a seguir.

Interface de programação - Estas interfaces permitem que usuários e administradores interajam com a plataforma de nuvem para utilizar suas funcionalidades. Estas interfaces permitem ao usuário desenvolver rotinas que manipulem os recursos contratados de acordo com a sua demanda. Por exemplo, uma rotina pode requisitar recursos adicionais caso verifique que os recursos contratados são insuficientes para atender a demanda atual. Existem diversas interfaces de programação como OpenStack API (*OpenStack Ap-*

plication Programming Interface) (FIFIELD et al., 2014), *Distributed Management Task Force* (DMTF) *Cloud Infrastructure Management Interface* (CIMI) (HARSH et al., 2012) e *Open Grid Forum* (OGF) *Open Cloud Computing Interface* (OCCI) (HARSH et al., 2012). Entretanto, a interface Amazon EC2 (VAN VLIET; PAGANELLI, 2011) é o padrão *de facto* e é adotada pela maioria das plataformas como OpenStack (FIFIELD et al., 2014), Eucalyptus (NURMI et al., 2009) e OpenNebula (TORALDO, 2012).

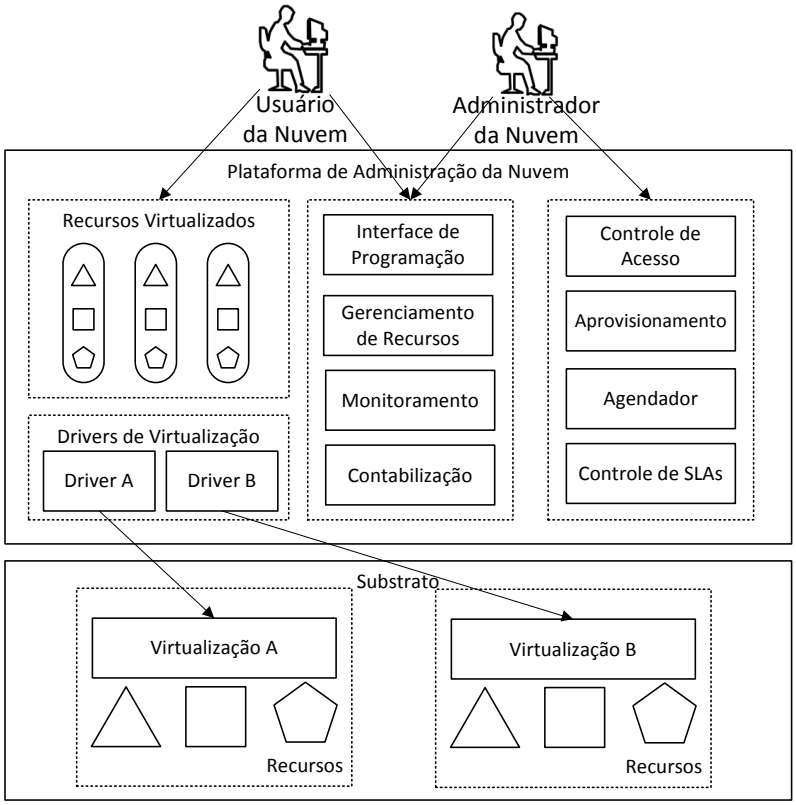


Figura 2.2: Arquitetura genérica de plataformas de administração da nuvem

Gerenciamento de recursos - Esta funcionalidade está relacionada principalmente à característica de elasticidade do paradigma de computação em nuvem. O gerenciamento de recursos permite que o usuário ajuste os recursos contratados conforme as suas necessidades (BUYYA; YEO; VENUGOPAL, 2008). Esta funcionalidade interage com os *drivers* de virtualização para ajustar os recursos de acordo com a requisição do usuário.

Monitoramento - Esta funcionalidade é importante para o funcionamento da plataforma, pois as informações coletadas pelo monitoramento são utilizadas por outras funções como contabilização, aprovisionamento e controle de SLAs. Além disso, o monitoramento fornece medições aos usuários e administradores em relação ao estado dos recursos contratados e do ambiente como um todo (MONTES et al., 2013).

Contabilização - Esta funcionalidade está relacionada ao modelo de cobrança por utilização adotado no paradigma de computação em nuvem. Esta funcionalidade pode tanto utilizar informações coletadas pelo monitoramento para permitir a cobrança de acordo com a utilização efetiva dos recursos como basear-se na reserva de recursos previamente realizada pelos usuários (BUYYA et al., 2009).

Controle de acesso - Esta funcionalidade está relacionada ao compartilhamento de recursos por múltiplos usuários. O administrador deve ser capaz de controlar quais recursos podem ser acessados por quais usuários (TAKABI; JOSHI; AHN, 2010).

Aprovisionamento - Esta funcionalidade auxilia os administradores no mapeamento entre os recursos virtualizados e os recursos físicos do substrato, ou seja, auxilia na escolha de quais recursos do substrato em que serão hospedados os recursos virtualizados. Este mapeamento será realizado de acordo com critérios estabelecidos pelos administradores (*e.g.* maior utilização de um servidor, balanceamento de carga, menor latência) (ESTEVES et al., 2013). Devido à dinamicidade do ambiente de computação em nuvem, este componente deve ser invocado periodicamente para que o mapeamento seja ajustado conforme o estado atual do ambiente (WICKBOLDT et al., 2014).

Agendador - O agendador é responsável pela criação dos recursos virtualizados no momento definido pelo usuário. Para tal, utiliza a função de aprovisionamento na escolha do recurso do substrato (ASSUNCAO; COSTANZO; BUYYA, 2009). Esta funcionalidade também é responsável por manter o mapeamento entre recursos virtuais e recursos do substrato, pois podem ocorrer migrações de recursos virtuais entre componentes do substrato para contornar falhas ou sobrecargas.

Controle de SLAs - O gerenciamento no âmbito dos serviços exige que SLAs sejam estabelecidos para o acompanhamento da qualidade de serviço dos recursos contratados. Esta funcionalidade utiliza as informações coletadas pelo monitoramento para analisar a conformidade entre a qualidade de serviço dos recursos acordada no SLA e aquela percebida pelos usuários (BASET, 2012).

Recursos virtuais - Estes recursos variam de acordo com o nível de abstração da nuvem. Em uma nuvem IaaS, estes recursos podem ser agrupados em uma máquina virtual ou em uma rede virtual com topologia definida pelo usuário conectando diversas máquinas virtuais (WICKBOLDT et al., 2014). Em uma nuvem PaaS, estes recursos podem ser servidores de aplicação ou plataformas de desenvolvimento. Durante esta dissertação utiliza-se o termo *cloud slice* para se referir aos recursos virtualizados independentemente se são máquinas virtuais, redes virtuais ou plataformas.

Drivers de virtualização - A plataforma de administração de nuvem não é diretamente responsável pela virtualização dos recursos do substrato. As plataformas utilizam *drivers* que são capazes de utilizar tecnologias de virtualização como Xen (BARHAM et al., 2003) e *Kernel-based Virtual Machine* (KVM) (KIVITY et al., 2007). Outra abordagem é utilizar um *driver* como o libvirt (BOLTE et al., 2010) que é capaz de interagir com diversas tecnologias de virtualização.

A seguir são apresentados exemplos de plataformas de administração de computação em nuvem que abrangem iniciativas da academia e da indústria. Posteriormente, são apresentados alguns exemplos dos serviços prestados por provedores de nuvens públicas.

Eucalyptus (NURMI et al., 2009) é uma plataforma de administração desenvolvida por iniciativa acadêmica para fomentar a pesquisa em computação em nuvem. Esta plataforma de administração permite a criação de ambientes para nuvens IaaS privadas e híbridas, pois possui compatibilidade com os serviços de nuvem pública da *Amazon Web Services* (AWS) (MURTY, 2008) (Amazon *Elastic Cloud Computing* (EC2), Amazon *Elastic Block Store* (EBS) e Amazon *Simple Storage Service* (S3)) (LONEA, 2013). Para auxiliar as tarefas de administração, a plataforma Eucalyptus fornece um conjunto de ferramentas para operações básicas chamado Euca2ools (LONEA, 2013). Como a interface de programação (API) utilizada pelo Eucalyptus é a Amazon EC2 (VAN VLIET; PAGANELLI, 2011), que é o padrão *de facto* para nuvens IaaS, o conjunto de ferramentas Euca2ools também pode ser utilizado para manipular recursos computacionais na nuvem IaaS da Amazon. O Eucalyptus possui *drivers* para as tecnologias de virtualização Xen, KVM e VMware ESX/ESXi (OGLESBY; HEROLD, 2005).

OpenNebula (TORALDO, 2012) (WEN et al., 2012) é uma plataforma desenvolvida por iniciativa acadêmica com o objetivo de fornecer uma plataforma de computação em nuvem baseada em componentes de código aberto. OpenNebula permite a criação de nuvens IaaS privadas e híbridas. A plataforma possui compatibilidade com interfaces de programação Amazon EC2, XML-RPC (LAURENT et al., 2001) e OGF OCCI. Os drivers de virtualização suportados são Xen, KVM, VMware ESX/ESXi/Server e libvirt.

OpenStack (PEPPLE, 2011) (SEFRAOUI; AISSAOUI; ELEULDJ, 2012) surgiu da iniciativa da NASA e da Rackspace Hosting de integrar e disponibilizar seus esforços em computação em nuvem como um projeto de código aberto. O principal objetivo do OpenStack é a escalabilidade da plataforma para nuvens IaaS. A plataforma possui uma interface de programação chamada OpenStack API (FIFIELD et al., 2014), e também é compatível com a Amazon EC2. OpenStack suporta diversos *drivers* de virtualização como Xen, KVM, VMware vSphere (LOWE, 2011), *User Mode Linux* (UML) (DIKE, 2006) e Hyper-V (VELTE; VELTE, 2009).

Aurora (WICKBOLDT et al., 2013) (WICKBOLDT et al., 2014) é uma plataforma de administração que está em desenvolvimento por iniciativa acadêmica no contexto do doutorado de Juliano Araujo Wickboldt do Grupo de Redes da UFRGS. O principal objetivo da plataforma de administração Aurora é introduzir flexibilidade na otimização e alocação de recursos através de programabilidade, bem como oferecer recursos e funcionalidades mais sofisticados voltados para redes dentro da nuvem adotando conceitos de redes virtuais (BARI et al., 2013) e *Software-defined Networking* (SDN) (KIM; FEAMSTER, 2013). Para oferecer estes recursos e funcionalidades, a plataforma permite que os usuários definam os recursos desejados e a topologia de rede virtual que os conecta através de um arquivo no formato *Virtual Resources and Interconnection Networks Description Language* (VXDL) (KOSLOVSKI et al., 2010). Aurora utiliza o libvirt como *driver* de virtualização, o que permite a utilização de diversas tecnologias de virtualização.

A Amazon é um dos provedores líderes para nuvens do tipo IaaS no mercado[1]. A Amazon oferece serviços de nuvem IaaS como a Amazon EC2, serviços de armazenamento Amazon S3 e Amazon EBS, bem como serviços de nuvem PaaS como Amazon *Elastic Load Balancing* (ELB) e Amazon *Relational Database Service* (RDS). Estes serviços são integrados à Amazon AWS. A Amazon desenvolveu uma interface de programação para a Amazon EC2 que tornou-se o padrão *de facto* para nuvens IaaS.

A Microsoft oferece serviços de computação em nuvem através da Windows Azure (TULLOCH, 2013). O serviço de nuvem IaaS permite a utilização de sistemas operacionais Windows e Linux. Existem também serviços como *Block Blobs* e *Page Blobs and Disks* para armazenamento de objetos e blocos, respectivamente. Na nuvem Windows Azure também são oferecidos serviços de nuvem PaaS para desenvolvimento de aplicações em .Net (RAJSHEKHAR, 2013), Python (LUTZ, 2013), Java (KNUDSEN; NIEMEYER, 2005), Ruby (FLANAGAN; MATSUMOTO, 2008) e *PHP: Hypertext Preprocessor* (PHP) (LERDORF; TATROE; MACINTYRE, 2006).

Rackspace Hosting disponibiliza serviços de nuvens públicas, privadas e híbridas. A Rackspace oferece serviços como *Cloud Servers*, *Block Storage* e *Cloud Files* de nuvem pública que são baseados no OpenStack. Também são oferecidos serviços de PaaS como *Cloud Databases*, *Cloud Domain Name System (DNS)* e *Load Balancers*.

[1]Fonte: *Gartner - Magic Quadrant for Cloud Infrastructure as a Service*, de 28 de maio de 2014.

2.3 Monitoramento de ambientes computacionais em nuvem

Os ambientes de computação em nuvem devem operar corretamente para que as organizações possam usufruir das características e benefícios do paradigma. Neste sentido, os administradores devem executar tarefas de gerenciamento para que o ambiente atenda às expectativas, como os SLAs estabelecidos e o lucro esperado. Entre as tarefas de gerenciamento, o monitoramento oferece aos administradores métricas que são fundamentais tanto para a operação quanto para o planejamento destes ambientes (FATEMA et al., 2014). Nesta seção são apresentadas as propriedades e peculiaridades que devem ser consideradas no monitoramento destes ambientes.

As peculiaridades dos ambientes de computação em nuvem exigem que as soluções de monitoramento possuam determinadas propriedades. Neste sentido, Aceto *et al.* (2013) realizaram um amplo estudo sobre as propriedades que soluções de monitoramento devem possuir para serem apropriadas para ambientes de computação em nuvem. Estas propriedades são apresentadas a seguir.

Escalabilidade - Esta propriedade está relacionada à quantidade de objetos e métricas que devem ser monitorados. Uma solução de monitoramento escalável deve possuir desempenho aceitável independentemente do número de objetos e métricas que são monitorados. Especialmente em ambientes de computação em nuvem que são naturalmente complexos envolvendo um grande número e variedade de dispositivos. Além da complexidade, a utilização de virtualização aumenta o número de objetos a serem monitorados, pois um único dispositivo pode hospedar diversos recursos virtualizados.

Elasticidade - Esta propriedade está relacionada ao dinamismo dos ambientes de computação em nuvem em que recursos virtuais são criados, ampliados, reduzidos e destruídos rapidamente para atender a demanda dos usuários. Além disso, a utilização de virtualização permite aos administradores do provedor migrar recursos virtuais entre recursos físicos para desligar equipamentos ociosos ou contornar falhas. Estas operações podem exigir que o monitoramento seja reconfigurado.

Resiliência, confiabilidade e disponibilidade - Estas propriedades estão relacionadas à importância do monitoramento para atividades críticas em ambientes de computação em nuvem, tais como tarifação, controle de SLAs e aprovisionamento de recursos. Logo, o monitoramento deve ser resiliente, ou seja, manter-se disponível na ocorrência de falhas na infraestrutura. Da mesma forma, o monitoramento deve ser confiável para que seus resultados possam ser utilizados nestas atividades críticas.

Adaptabilidade - Esta propriedade também está relacionada ao dinamismo dos ambientes de computação em nuvem, pois o monitoramento deve ser sensível à carga do ambiente adaptando-se para que não seja intrusivo e prejudique o desempenho do ambiente. A adaptação pode ocorrer tanto pela diminuição da frequência das medições quanto pelo ajuste de filtros que limitam a quantidade de medições que são enviadas à solução de monitoramento (CLAYMAN et al., 2011). Entretanto, perde-se precisão ao filtrar medições ou diminuir sua frequência, portanto é necessário ajustar esta capacidade de adaptação de acordo com requisitos de desempenho juntamente com requisitos de precisão e de escalabilidade (RODRIGUES et al., 2014).

Oportunidade - Esta propriedade está relacionada à capacidade do monitoramento de fornecer informações no intervalo de tempo oportuno, ou seja, no momento em que as informações são úteis e relevantes. Estas informações são necessárias para a operação e funcionamento do ambiente. Por exemplo, o monitoramento deve disparar alarmes logo após a ocorrência de uma falha para que as medidas sejam tomadas rapidamente para contorná-la e não afetar SLAs. Além de detectar falhas, o monitoramento fornece infor-

mações sobre a utilização da infraestrutura que são necessárias para o aprovisionamento eficiente dos recursos de acordo com a demanda atual.

Autonomicidade - Esta propriedade está relacionada à capacidade do sistema de monitoramento funcionar sem intervenções dos administradores. Portanto, o sistema de monitoramento deve ser capaz de detectar e, se necessário, ajustar a sua configuração para adaptar-se às mudanças de forma autônoma. Esta propriedade é importante porque ambientes de computação em nuvem oferecem dinamismo aos usuários que podem criar, modificar e destruir os recursos contratados a qualquer tempo sem uma comunicação aos administradores do ambiente. Além disso, os mecanismos de aprovisionamento das plataformas de administração da nuvem tomam decisões de forma autônoma, o que pode ocasionar migrações de recursos virtuais para o melhor aproveitamento da infraestrutura. Desta forma, os administradores não devem estar envolvidos na reconfiguração do monitoramento para refletir as operações que foram realizadas pelos usuários, pelos administradores ou pelas plataformas de administração da nuvem de forma autônoma.

Abrangência, extensibilidade e baixa intrusividade - Abrangência está relacionada à capacidade da solução de monitoramento suportar diferentes tipos de recursos. Por exemplo, a solução de monitoramento deve ser capaz de monitorar tanto os recursos da infraestrutura como aqueles disponibilizados aos usuários. Extensibilidade está relacionada à possibilidade de adicionar funcionalidades à solução de maneira que seja capaz de monitorar outros tipos de recursos. Intrusividade está relacionada à quantidade de modificações que devem ser realizadas no ambiente de computação em nuvem para que o monitoramento funcione adequadamente. Logo, espera-se uma baixa intrusividade para que a implantação das soluções de monitoramento seja mais fácil.

Precisão - Esta propriedade está relacionada à qualidade das medições que a solução de monitoramento é capaz de oferecer. Medições precisas são fundamentais em ambientes de computação em nuvem, pois os valores medidos são utilizados em tarefas críticas como tarifação e aprovisionamento de recursos. A precisão também está associada à frequência em que as medições são realizadas. Quanto maior o número de medições em um determinado período de tempo, maior poderá ser a precisão destas medições. Entretanto, também será maior a influência do monitoramento em relação à utilização de banda e consumo de recursos, o que pode prejudicar o desempenho do ambiente. Desta forma, há um compromisso entre a precisão e a intrusividade do monitoramento.

Federação - Esta propriedade está relacionada à capacidade da solução de monitorar recursos que se encontram em outros domínios (CLAYMAN et al., 2010). Esta propriedade é necessária quando um usuário utiliza recursos de diversas nuvens públicas ou de nuvens híbridas. Nestes casos, o monitoramento não deve ser afetado por migrações de recursos entre nuvens públicas ou da nuvem privada para a pública e vice-versa.

Além de exigir propriedades específicas, o monitoramento de ambientes computacionais em nuvem está inserido em um cenário bastante distinto em relação ao monitoramento de ambientes tradicionais. A Figura 2.3 (MONTES et al., 2013) ilustra este cenário em duas dimensões: o nível e a visão. O nível está relacionado a qual camada do ambiente pertence o objeto a ser monitorado, desde o sistema físico até o sistema virtual. Além disso, o sistema virtual é subdividido em camadas de acordo com o nível de abstração dos serviços oferecidos desde infraestruturas (IaaS), plataformas (PaaS) até aplicações (SaaS). Na outra dimensão, a visão está relacionada ao público alvo das medições oferecidas pelo monitoramento. Pela visão dos clientes, as medições relacionadas ao sistema virtual são interessantes para o acompanhamento dos serviços contratados. Entretanto, pela visão do provedor as medições do sistema virtual são interessantes para acompanhar a qualidade

do serviço do usuário e garantir que estes serviços estejam de acordo com os SLAs estabelecidos. Já as medições do sistema físico auxiliam na operação do ambiente e são fundamentais para tarefas como aprovisionamento.

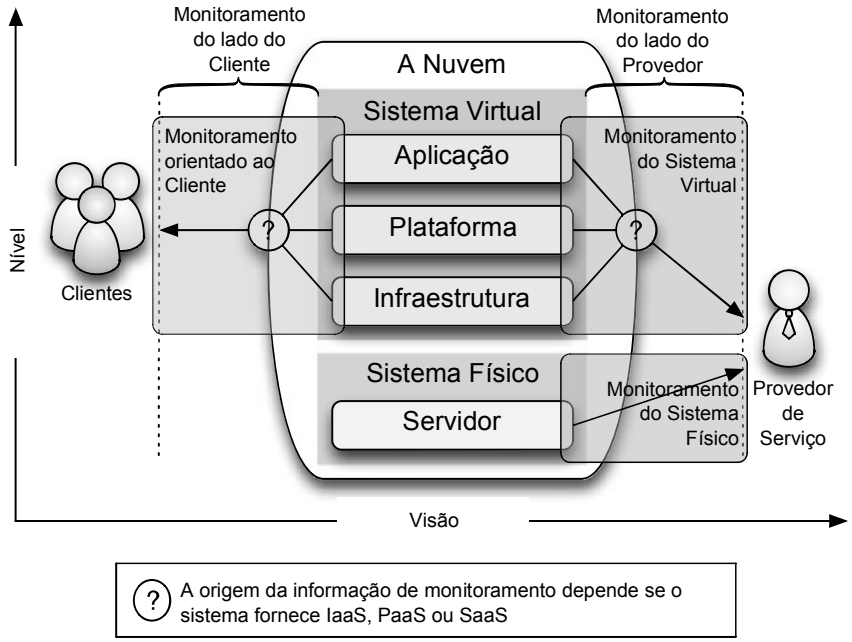


Figura 2.3: Cenário típico do monitoramento de ambientes computacionais em nuvem (traduzido de Montes *et al.* (2013))

2.4 Soluções de monitoramento

Existe um grande número de soluções de monitoramento para ambientes computacionais em nuvem tanto propostas pela academia como desenvolvidas pela indústria e comunidades de software livre, o que impede que uma lista exaustiva de soluções seja apresentada. Por este motivo, nesta seção são apresentadas soluções de monitoramento escolhidas por serem as mais relevantes para o contexto desta dissertação. Estas soluções diferem quanto ao nível dos objetos que monitoram e à visão que proporcionam. Além disso, estas soluções também diferem quanto à capacidade de atender às propriedades desejadas para o monitoramento de ambientes computacionais em nuvem. Neste sentido, soluções de monitoramento são apresentadas com os níveis dos objetos que monitoram e a visão que proporcionam. Ao final, a Tabela 2.1 apresenta as propriedades que cada uma das soluções apresentadas é capaz de atender.

GMonE (MONTES et al., 2013) é uma solução de monitoramento modular desenvolvida para todos níveis e visões do cenário ilustrado pela Figura 2.3. A solução utiliza o paradigma de comunicação *publish-subscribe* para enviar medições de monitoramento dos módulos *GMonEMon* para serem armazenadas pelos módulos *GMonEDB*. Os módulos *GMonEMon* são responsáveis pela coleta das medições e podem ser estendidos através de *plugins* desenvolvidos em Java (KNUDSEN; NIEMEYER, 2005). O módulo *GMonEAccess* permite acesso aos dados armazenados nos *GMonEDB* de acordo com a visão apropriada ao usuário. Embora a solução possa ser estendida por *plugins*, a sua extensibilidade é dificultada por exigir que eles sejam desenvolvidos em Java. Sem esta restrição, *plugins* de outras soluções de monitoramento poderiam ser utilizados.

CloudWatch (VAN VLIET; PAGANELLI, 2011) é a solução de MaaS que oferece a visão do usuário em todos os níveis do sistema virtual (Figura 2.3) para os serviços da Amazon e para usuários da plataforma Eucalyptus. No nível *Infrastructure* oferece medições sobre os recursos computacionais de instâncias hospedadas na Amazon EC2. No nível *Platform* oferece medições sobre os serviços de balanceamento de carga (ELB) e de base de dados relacionais (RDS) da Amazon. Adicionalmente, as aplicações desenvolvidas pelos clientes podem publicar medições personalizadas no CloudWatch, o que permite monitoramento do nível *Application*. Entretanto, pode-se concluir a partir da documentação disponível que a solução apresenta problemas quanto à adaptabilidade, informações no tempo oportuno e precisão. Estas propriedades estão diretamente relacionadas à frequência das medições, que podem ser de 1 ou 5 minutos, de acordo com a assinatura. Estas medições não possuem uma precisão apropriada principalmente porque são utilizadas pelo serviço de balanceamento de carga.

O mOSAIC *framework* (RAK et al., 2011) permite que desenvolvedores criem soluções de monitoramento para suas aplicações. Estas soluções são construídas integrando componentes oferecidos pelo *framework* que realizam a coleta de informações de monitoramento desde a infraestrutura até a aplicação. As informações coletadas são compartilhadas através de barramentos de eventos, por exemplo, para agrupar todos os eventos relacionados à aplicação. As soluções de monitoramento criadas devem ser alteradas manualmente para adequar-se às modificações no ambiente da nuvem. Desta forma, a escalabilidade, elasticidade e autonomicidade destas soluções ficam prejudicadas.

Dargos (CORRADI et al., 2012) é um sistema de monitoramento centrado nos dados (*data-centric*) (PARDO-CASTELLOTE; FARABAUGH; WARREN, 2005), ou seja, a lógica da aplicação é baseada nos dados e não nas entidades que geram ou recebem os dados. Neste sentido, Dargos utiliza um sistema chamado *Data Distribution Service* (DDS) para disponibilizar os dados de monitoramento. O sistema DDS é baseado no paradigma *publish/subscribe* em que os agentes (*publishers*) coletam dados de monitoramento dos recursos computacionais. Estes dados são enviados para entidades (*subscribers*) que registraram seu interesse nos dados de monitoramento. As entidades interessadas podem ser quaisquer módulos ou sistemas que dependam de dados do monitoramento para seu funcionamento como, por exemplo, módulos para aprovisionamento de recursos, agendadores ou interfaces de monitoramento e de administração.

PCMONS (DE CHAVES; URIARTE; WESTPHALL, 2011) é um sistema cujo objetivo é prover monitoramento para nuvens privadas. Sua arquitetura é baseada em três camadas: *Infrastructure*, *Integration* e *View*. A camada *Infrastructure* abriga a heterogeneidade do ambiente de computação em nuvem, como as plataformas (*e.g.* Eucalyptus, OpenNebula) e tecnologias de virtualização (*e.g.* Xen, KVM). A camada *Integration* oferece uma interface comum de acesso às informações e abstrai a heterogeneidade da camada *Infrastructure*. A camada *View* utiliza esta interface comum para oferecer visualização apropriada de acordo com o público alvo do monitoramento (Visão).

Lattice (CLAYMAN; GALIS; MAMATAS, 2010) é um *framework* que permite a construção de sistemas de monitoramento para redes virtuais, cujos requisitos são semelhantes aos de monitoramento de computação em nuvem (*e.g.* escalabilidade, elasticidade, adaptabilidade e autonomicidade). Lattice utiliza o paradigma de produtores e consumidores para distribuir informações de monitoramento coletadas pelos *probes* (produtores) para os sistemas de monitoramento (consumidores). Esta distribuição pode ser realizada por mecanismos como *Internet Protocol* (IP) *Multicast*, barramento de eventos ou abordagem *publish/subscribe*.

Monalytics (KUTARE et al., 2010) é um sistema que integra monitoramento e análise das informações coletadas. O monitoramento e análise é realizado por agentes próximos aos objetos a serem monitorados afim de aumentar a escalabilidade do sistema. As informações coletadas são enviadas para *brokers* que tomam decisões baseadas na análise dos dados coletados. Os *brokers* podem ajustar o monitoramento para uma determinada situação, por exemplo, através da alteração da taxa de amostragem para obter mais precisão ou para ser menos intrusivo. Os *brokers* também podem disparar ações para contornar falhas, por exemplo, através da reinicialização de um serviço ou máquina virtual.

TIMaCS (VOLK et al., 2012) é um sistema que integra funcionalidades de monitoramento e gerenciamento separadas em dois blocos. No bloco de monitoramento são executadas funções como coleta, agregação, armazenamento, testes de conformidade, testes de regressão (para previsão de problemas) e filtragem/geração de eventos. No bloco de gerenciamento são executadas funções como base de conhecimento, controlador para disparar comandos em outros nodos ou em delegados de recursos, tratamento de eventos, tomador de decisões e um módulo para receber comandos de outros nodos. A integração de todas estas funções permite aos administradores que comandos sejam executados na ocorrência de determinados eventos. Desta forma, o monitoramento fornece medições que são utilizadas para a tomada das decisões necessárias para a execução de comandos.

Além destas, outras soluções de monitoramento também podem ser empregadas no contexto de computação em nuvem. Ceilometer (FIFIELD et al., 2014) é um *framework* para o monitoramento centralizado de todos componentes de um ambiente de nuvem baseado no OpenStack. Seu objetivo é centralizar resultados de medições para tarefas como tarifação e aprovisionamento de recursos. RMCM (*Runtime Model for Cloud Monitoring*) (SHAO et al., 2010) é uma solução de monitoramento que integra medições de todos os níveis da nuvem referentes a aplicações Java. O RMCM utiliza técnicas como instrumentação para obter medições diretamente da aplicação, interceptadores para capturar informações sobre as requisições que estão sendo processadas e utiliza bibliotecas nativas para capturar informações das máquinas virtuais e físicas.

O monitoramento de ambientes computacionais em nuvem também pode ser realizado através de soluções tradicionais de monitoramento como Nagios (KOCJAN, 2014), Zabbix (VACCHE; LEE, 2013) e MRTG (SHIPWAY; OETIKER, 2010). Estas soluções são utilizadas em ambientes heterogêneos e podem monitorar recursos implementados em ambientes de computação em nuvem. Por exemplo, o Nagios possui *plugins* para o monitoramento de banco de dados, servidores de aplicação, plataformas de virtualização e recursos físicos como processamento, memória e armazenamento. Estas ferramentas são amplamente utilizadas, possuem farta documentação e administradores estão acostumados a utilizá-las. Entretanto, elas não são integradas às plataformas de computação em nuvem e não suportam a dinamicidade destes ambientes.

MonPaaS (CALERO; AGUADO, 2015) é uma plataforma que integra a solução de monitoramento Nagios à plataforma de computação em nuvem OpenStack. MonPaaS configura o Nagios automaticamente quando novos *cloud slices* são criados no OpenStack, bem como desativa o monitoramento quando estes *cloud slices* são deletados pelos usuários. Para descobrir a criação e deleção de *cloud slices* no OpenStack, MonPaaS observa as comunicações entre os componentes da plataforma diretamente no barramento de mensagens do OpenStack em busca das mensagens necessárias para a criação e deleção dos *cloud slices*. Para configurar o Nagios, MonPaaS realiza chamadas para o *REpresentational State Transfer* (REST) *Web Service* do NConf (SIMÕES, 2010), que é uma interface de gerenciamento para o Nagios. MonPaaS também utiliza o DNX (JOSEPH-

SEN, 2013) que permite a criação de *clusters* onde um servidor Nagios mestre comanda diversos servidores Nagios escravos para atingir maior escalabilidade. Entretanto, o MonPaaS possui alto acoplamento com o Nagios e OpenStack, o que limita a sua adoção em ambientes com outras soluções de monitoramento e plataformas de nuvem. Além disso, a solução cria uma *Monitoring Virtual Machine* (MVM) para cada usuário da nuvem para agrupar todos os serviços de monitoramento daquele usuário, o que aumenta a utilização de recursos da nuvem que serão destinados exclusivamente para o monitoramento.

A Tabela 2.1 apresenta as propriedades atendidas pelas soluções de monitoramento apresentadas nesta seção. Para determinar se a solução de monitoramento atende a uma propriedade, foi realizada pesquisa nos trabalhos que apresentam estas soluções em busca de referências às propriedades ou suas características. Uma pesquisa semelhante foi realizada na documentação das soluções fornecidas pela indústria como Nagios, Zabbix e MRTG. Logo, se determinada propriedade não está assinalada para uma solução, significa que a solução não atende àquela propriedade ou não há indícios suficientes nos trabalhos e documentações que assegurem que a solução atende àquela propriedade. Pode-se perceber na Tabela 2.1 que a propriedade federação não está assinalada para nenhuma solução, embora esteja prevista como trabalho futuro na implementação do Lattice. Outras propriedades como resiliência, confiabilidade, disponibilidade e precisão não estão assinaladas para nenhuma solução, pois estas propriedades são inerentemente percebidas na utilização das soluções em ambientes reais ou de avaliação, o que não foi realizado nesta dissertação.

Tabela 2.1: Propriedades oferecidas pelas soluções de monitoramento

Solução	**Escalabilidade**	**Elasticidade**	**Resiliência**	**Confiabilidade**	**Disponibilidade**	**Adaptabilidade**	**Oportunidade**	**Autonomicidade**	**Abrangência**	**Extensibilidade**	**Baixa intrusividade**	**Federação**
GMonE	✓									✓	✓	
CloudWatch		✓						✓		✓		
mOSAIC										✓		
Dargos						✓				✓	✓	
PCMONS										✓		
Lattice		✓						✓		✓		
Monalytics	✓	✓				✓	✓					
TIMaCS	✓								✓	✓		
Ceilometer								✓		✓		
RMCM									✓	✓	✓	
Nagios									✓	✓		
Zabbix									✓	✓		
MRTG									✓			
MonPaaS	✓	✓						✓	✓	✓		

2.5 Composição de soluções de monitoramento e automação de suas configurações

Nesta seção são discutidas as possíveis abordagens para ampliar os requisitos atendidos pelas soluções de monitoramento, bem como é discutida a necessidade de configuração automática destas soluções para computação em nuvem. Também é discutido porque soluções de automação de uso geral não são apropriadas para a configuração de soluções de monitoramento para computação em nuvem.

Aceto *et al.* (2013) realizaram um estudo sobre soluções de monitoramento para computação em nuvem e as propriedades que estas soluções são capazes de atender. Neste estudo, um grande número de soluções foi avaliado e apresentado como na Tabela 2.1, o que permite concluir que atualmente nenhuma solução de monitoramento atende a todas propriedades. Portanto, é necessário aplicar novas abordagens para ampliar as propriedades satisfeitas por soluções de monitoramento. Por exemplo, pode-se utilizar uma única solução de monitoramento desenvolvida ou adaptada de outra existente de modo que atenda a todas as propriedades necessárias; ou compor múltiplas soluções para construir um sistema de monitoramento que agrega as propriedades atendidas por cada solução.

O desenvolvimento de uma nova solução exigiria um grande esforço para alcançar todas as propriedades citadas anteriormente. Da mesma forma, a adaptação de uma solução existente também exigiria um grande esforço porque o cenário de computação em nuvem é bastante diferenciado conforme ilustrado na Figura 2.3 e ainda dependeria da capacidade de adaptação da solução de monitoramento existente. Além disso, em ambientes tradicionais os administradores costumam utilizar diversas soluções de monitoramento para explorar as funcionalidades de cada uma. Por exemplo, um administrador pode utilizar o Nagios (KOCJAN, 2014) para monitorar o estado de objetos e utilizar o MRTG (SHIPWAY; OETIKER, 2010) para produzir gráficos de utilização de recursos. Portanto, pode-se esperar que mesmo com soluções de monitoramento mais completas os administradores ainda utilizariam outras soluções para explorar funcionalidades específicas.

A composição de soluções de monitoramento não pode ser descartada, pois administradores costumam utilizar múltiplas soluções de monitoramento. Entretanto, quanto maior o número de soluções adotadas, maior será a complexidade da configuração do monitoramento. Além disso, os ambientes de computação em nuvem são naturalmente dinâmicos e algumas operações realizadas no ambiente devem ser refletidas no monitoramento, tais como a criação de máquinas virtuais em uma nuvem IaaS. Logo, espera-se que as soluções de monitoramento acompanhem estas operações de maneira autônoma, ou seja, sem a intervenção de um administrador.

Existem soluções de monitoramento que são integradas às plataformas de computação em nuvem como Ceilometer e CloudWatch, e portanto são automaticamente configuradas quando operações são realizadas no ambiente de computação em nuvem. Entretanto, existem soluções que não são integradas e exigem intervenção manual ou utilizam um processo de descoberta para serem configuradas. Durante o processo de descoberta, a rede é escaneada em busca de novos dispositivos que devem ser monitorados. Entretanto, esta abordagem não é apropriada para ambientes dinâmicos, como o de computação em nuvem, pois exigiria que este processo fosse executado frequentemente consumindo potencialmente uma grande quantidade de recursos.

Outra abordagem possível para obter autonomicidade na configuração das soluções de monitoramento seria utilizar ferramentas de automação como Puppet (FRANCESCHI, 2014) ou Chef (MARSCHALL, 2013). No Puppet, um servidor armazena as configura-

ções que devem ser aplicadas nos clientes na forma de manifestos e classes. Cada classe representa um tipo de serviço e possui atributos predefinidos relacionados à configuração do serviço. Nos manifestos os clientes (nodos) são declarados e são definidas quais configurações estes clientes devem receber. Estes manifestos são compilados para gerar um catálogo de configurações para todos os nodos da rede. Um agente instalado nos clientes recupera as configurações que devem ser aplicadas comunicando-se com o catálogo. Logo, as configurações são distribuídas a partir do servidor através de um mecanismo *pull* em que os clientes buscam as suas configurações em um determinado intervalo de tempo.

O Chef utiliza receitas escritas em Ruby que determinam todos os passos necessários para que uma configuração seja aplicada a um nodo. As receitas são armazenadas em *cookbooks* que utilizam outros componentes do Chef como *templates*, bibliotecas e atributos dos nodos. Além de receitas, os nodos podem receber papéis que determinam os serviços que serão instalados e configurados. Por exemplo, se um nodo recebe o papel de ser um servidor *Web*, o serviço Apache (LAURIE; LAURIE, 2003) é automaticamente instalado e configurado. Para cada execução do cliente Chef, uma *run-list* é gerada baseada nas receitas e nos papéis atribuídos ao nodo. As *run-lists* representam todas as ações que deverão ser executadas no nodo para que suas configurações e serviços atinjam o estado desejado pelo administrador. Estas *run-lists* são armazenadas no servidor Chef para serem comparadas com *run-lists* geradas futuramente e avaliar se ocorreram mudanças que exijam reconfiguração e evitar que receitas e papéis sejam reaplicados.

Puppet e Chef são soluções para a automação de tarefas e configurações nos nodos clientes. Entretanto, além da configuração nos clientes que serão monitorados, o monitoramento exige tarefas de configuração no servidor de monitoramento. O Chef possibilita a configuração do servidor de monitoramento através de *templates* que podem utilizar informações do catálogo. Por exemplo, a lista de todos os clientes registrados no Chef poderia gerar uma configuração com todos os nodos que devem ser monitorados pelo servidor de monitoramento. Entretanto, para distribuir o monitoramento entre diversos servidores seria necessário criar vários *templates* utilizando alguma lógica para dividir os nodos entre cada *template* para evitar conflitos. Além disso, os modelos de informação adotados pelas soluções de automação são inflexíveis e de uso geral o que exige a alteração de classes e *cookbooks* para a incorporação de novos atributos.

No decorrer deste capítulo foram descritas as propriedades necessárias em uma solução de monitoramento para ambientes de computação em nuvem. Entretanto, não existe uma única solução capaz de atender a todas as propriedades. Por outro lado, o desenvolvimento ou adaptação de uma solução que contemple estas propriedades pode ser muito complexo. Estes fatores evidenciam que a utilização de múltiplas soluções de monitoramento possa ser uma alternativa para o monitoramento de ambientes de computação em nuvem. Desta forma, pode-se ampliar a quantidade de propriedades atendidas e funcionalidades disponíveis para o monitoramento. Entretanto, existem soluções que não são integradas às plataformas de computação em nuvem, o que exige soluções de automação para contornar a falta de dinamicidade e autonomicidade que a integração proporcionaria. No próximo capítulo é apresentada uma solução que permite a utilização de múltiplas soluções de monitoramento e automatiza a configuração daquelas soluções que não são integradas às plataformas de computação em nuvem.

3 SOLUÇÃO PROPOSTA

Neste capítulo é apresentado o *framework* proposto e implementado nesta dissertação chamado FlexACMS (*Flexible Automated Cloud Monitoring Slices*). O conceito de *monitoring slice* foi introduzido no contexto desta dissertação e reflete todas as informações de monitoramento sobre um *cloud slice*. Ou seja, os *monitoring slices* são compostos pelas métricas monitoradas e as configurações necessárias para coletá-las (CARVALHO et al., 2013). Os *monitoring slices* são construídos utilizando diversas soluções para atender a um maior número de propriedades e funcionalidades, pois uma única solução de monitoramento não é capaz de atender a todas necessidades dos administradores. Na Figura 3.1 são ilustrados exemplos de *cloud slices* e seus respectivos *monitoring slices*, cujas métricas são coletadas por soluções projetadas para computação em nuvem, como Ceilometer (FIFIELD et al., 2014), e por ferramentas tradicionais de monitoramento, como Nagios (KOCJAN, 2014) e MRTG (SHIPWAY; OETIKER, 2010).

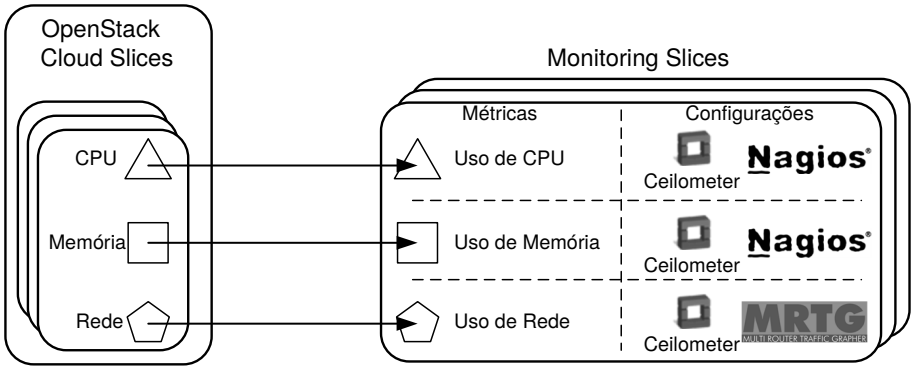


Figura 3.1: *Monitoring slices* que utilizam tanto soluções de monitoramento tradicionais como soluções específicas para computação em nuvem

Para atingir satisfatoriamente os objetivos propostos nesta dissertação, o FlexACMS possui determinadas características tais como: o baixo acoplamento entre o *framework* e soluções de monitoramento, bem como baixo acoplamento com plataformas de computação em nuvem; e a utilização de um modelo de informação flexível. O baixo acoplamento exige que a solução não dependa de funcionalidades específicas de soluções de monitoramento e plataformas de computação em nuvem. Desta forma, o baixo acoplamento é importante para que o administrador consiga escolher com liberdade qualquer solução de monitoramento para qualquer plataforma de computação em nuvem. Esta liberdade é reforçada por um modelo de informação flexível, pois este modelo é capaz de armazenar qualquer tipo de informação disponível nas plataformas e até mesmo armazenar informações complementares que podem ser fornecidas pelos administradores.

O restante deste capítulo é organizado como segue. Na Seção 3.1 é apresentado o modelo de informação que foi criado para a solução. Nas Seções 3.2 e 3.3 são apresentadas as arquiteturas inicial e estendida do FlexACMS, respectivamente, juntamente com as funções de cada componente destas arquiteturas. Na Seção 3.3 também são discutidas as diferenças entres as duas versões do FlexACMS, os benefícios e as funcionalidades adicionais que a arquitetura estendida introduz ao *framework*.

3.1 Modelo de Informação

Nesta seção é apresentado o modelo de informação desenvolvido para ser utilizado no FlexACMS. A Figura 3.2 ilustra o modelo de informação proposto para o FlexACMS. Este modelo é hierárquico e composto por quatro níveis: *Platform*, *Cloud*, *Slice*[1] e *Resource*, que armazenam informações sobre plataformas, nuvens, *cloud slices* e recursos, respectivamente. No contexto do *framework*, uma plataforma é definida como um conjunto de nuvens. Uma nuvem é um conjunto de *cloud slices* que compartilham alguma característica, por exemplo, pertencem ao mesmo usuário. Um *cloud slice* é um conjunto de recursos. Os recursos são a menor unidade que pode ser representada e referem-se aos recursos computacionais que compõem um *cloud slice* como, por exemplo, capacidades de processamento, de memória e de armazenamento.

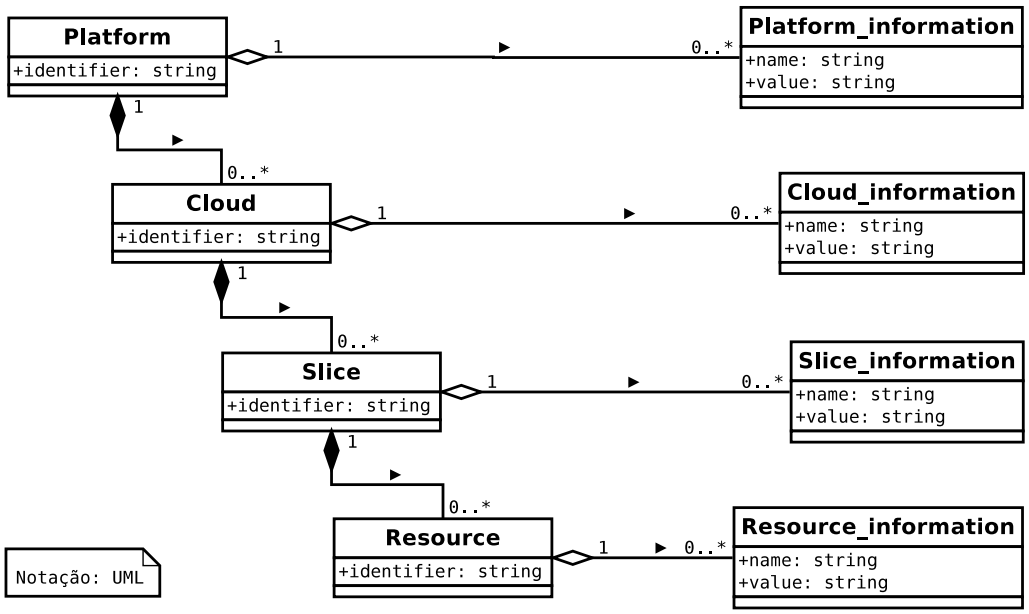


Figura 3.2: Modelo de informação hierárquico para representar *Platforms*, *Clouds*, *Slices*, *Resources* e seus atributos associados

O modelo de informação proposto é hierárquico para facilitar a seleção de atributos referentes ao objeto de um nível, pois o caminho realizado para chegar ao nível do objeto a partir do topo (*Platform*) é único e por consequência os objetos que são membros dos níveis deste caminho são unicamente identificados. Por exemplo, quando seleciona-se um objeto do nível *Slice* do modelo, tem-se também os respectivos objetos dos níveis *Cloud* e *Platform* unicamente identificados. Desta forma, um objeto de um nível do modelo

[1]Para simplificar, o termo *"Slice"* é utilizado quando refere-se ao nível do modelo de informação que se refere aos *cloud slices*

(*Platform*, *Cloud*, *Slice* ou *Resource*) define um escopo, de onde se pode acessar qualquer atributo do próprio objeto e dos objetos dos níveis superiores sem a necessidade de identificar diretamente o objeto do nível que se está interessado.

O significado de cada nível do modelo é genérico do ponto de vista do *framework*. Cabe ao administrador que utiliza o *framework* atribuir um significado relevante para cada nível do modelo. Por exemplo, para o *framework* uma nuvem é um conjunto de *cloud slices* que compartilham alguma característica. Logo, uma organização poderia utilizar o nível *Cloud* do modelo para distinguir *cloud slices* em fases diferentes, como desenvolvimento ou produção. Desta forma, o nível *Cloud* teria duas nuvens: desenvolvimento e produção. O mesmo aplica-se para *cloud slices* que são conjuntos de recursos para o *framework*. Para uma determinada organização, os *cloud slices* podem ser conjuntos de recursos de baixo nível de uma máquina virtual, tais como processamento, memória e armazenamento. Porém, para outra organização os *cloud slices* podem ser conjuntos de recursos de alto nível como a topologia de uma rede virtual. Neste caso, os recursos seriam roteadores, *switches*, nodos e interfaces.

A única informação obrigatória para cada nível é seu identificador (`identifier`). As demais informações são armazenadas nas estruturas auxiliares: *Platform_information*, *Cloud_information*, *Slice_information* e *Resource_information* que armazenam informações sobre *Platform*, *Cloud*, *Slice* e *Resource*, respectivamente. Cada uma destas estruturas auxiliares comporta-se como uma *Key Value Database* (CARLSON, 2013) para o objeto do nível associado, onde `name` é o identificador da informação que está armazenada e *value* é o valor armazenado. Por este motivo, o modelo de informação é flexível, pois não limita o conjunto de informações que podem ser representadas e também não exige um conjunto mínimo de informações que devem ser armazenadas.

Existem propostas de modelos e formatos para a representação de informações sobre cenários de computação em nuvem e de recursos virtualizados. Dentre estas propostas pode-se destacar o DMTF *Open Virtualization Format* (OVF) (HARSH et al., 2012), o OGF OCCI (*Open Cloud Computing Interface*) e o VXDL. O DMTF OVF é uma proposta de padronização de formato para a troca de informações na distribuição de softwares que serão executados em sistemas virtuais. Dentre estas informações pode-se representar atributos relacionados aos recursos virtuais das máquinas virtuais que executarão estes softwares. O OGF OCCI possui uma especificação chamada OCCI *Infrastructure* (HARSH et al., 2012) que representa exclusivamente informações sobre recursos virtuais. O VXDL surgiu de uma proposta acadêmica para representar topologias de redes virtuais juntamente com seus recursos virtualizados.

Um modelo de informação foi desenvolvido para o FlexACMS, embora existam propostas de padronização para modelos de informação. A principal motivação para o desenvolvimento de um novo modelo é a generalidade requerida para o *framework* e a consequente flexibilidade exigida no modelo de informação, que deve ser capaz de representar qualquer tipo de informação sobre os recursos virtuais. Embora as propostas citadas anteriormente sejam extensíveis, seria necessário um grande esforço de aprendizado destas propostas por parte dos administradores e desenvolvedores que optem por utilizar o *framework*. O esforço de aprendizado do modelo de informação desenvolvido para o FlexACMS é muito menor comparado ao esforço de aprendizado das propostas de padronização. Além disso, ainda não há uma clara intenção das soluções atuais de computação em nuvem em favor de um modelo ou formato para representar suas informações. Logo, caso o *framework* optasse por alguma destas propostas, os administradores de soluções de computação em nuvem que adotam outras propostas seriam penalizados.

Na solução proposta, o modelo de informação é fundamental porque influencia diretamente na generalidade do *framework* proposto. Por exemplo, um modelo de informação inflexível pode não ser capaz de representar determinadas informações, o que impediria a utilização do *framework* em cenários onde estas informações sejam necessárias. O modelo de informação deve ser flexível para permitir que informações disponíveis nas diversas plataformas de computação em nuvem possam ser utilizadas no *framework*. Além disso, um modelo flexível permite que administradores de nuvem utilizem atributos personalizados relacionados aos serviços contratados pelos usuários. Por exemplo, o *framework* pode ter informações relacionadas às assinaturas de serviços do tipo *Monitoring as a Service* (MaaS) contratadas pelos usuários, para que o monitoramento seja configurado de acordo com o que foi contratado.

A Figura 3.3 apresenta uma possível instância do modelo de informação desenvolvido para o FlexACMS. Esta instância é apenas um exemplo, visto que o significado de cada nível do modelo pode ser modificado pelo administrador. Neste exemplo, o modelo de informação representa informações sobre nuvens hospedadas na plataforma OpenStack. Estas nuvens representam conjuntos de *cloud slices* em diferentes fases: desenvolvimento, homologação e produção. Os *cloud slices* são máquinas virtuais compostas pelos recursos: *Central Processing Unit* (CPU), memória e disco. Os objetos com fundo cinza pertencem ao caminho único do escopo definido pelo *Slice vmprod02*. Por simplicidade, foram ilustradas apenas as estruturas auxiliares para os objetos que pertencem a este caminho, cujas informações estão representadas pelas tabelas *Platform_information*, *Cloud_information* e *Slice_information*. Em virtude do escopo, estas informações podem ser acessadas diretamente sem a necessidade de identificação direta dos seus objetos. Por exemplo, no escopo do *Slice vmprod02* pode-se acessar os atributos `@slice.ip`, `@cloud.schedule` e `@platform.API` diretamente sem a necessidade de identificar os objetos dos níveis *Slice*, *Cloud* e *Platform*, respectivamente, a qual se referem os atributos. Estes atributos podem ser utilizados pelos *Configurators* para a verificação de condições e também podem ser utilizados como argumentos para a configuração das soluções de monitoramento, conforme é apresentado nas próximas seções deste capítulo.

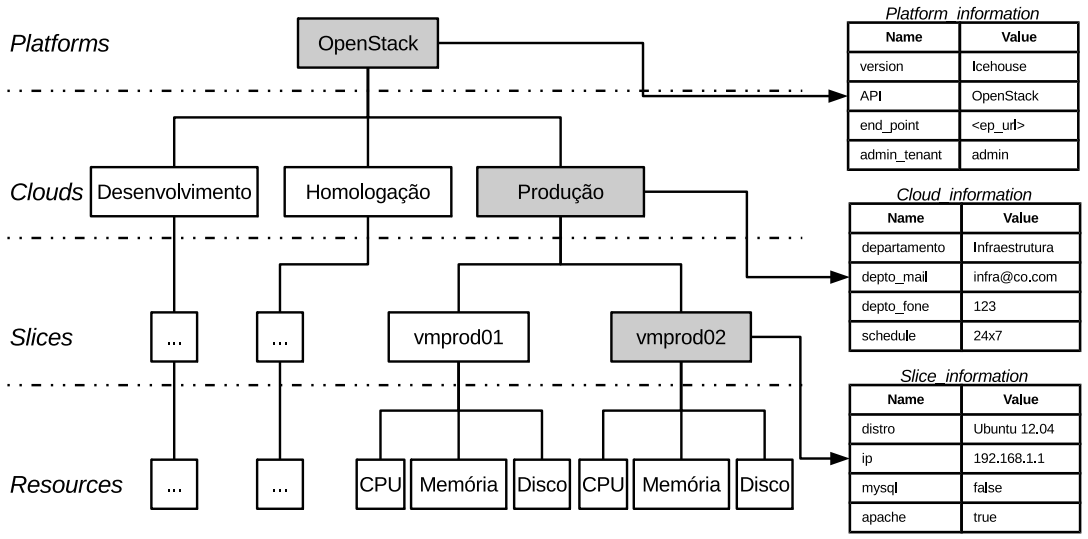


Figura 3.3: Exemplo de instância do modelo de informação hierárquico para uma organização que utiliza a plataforma OpenStack para hospedar nuvens de desenvolvimento, homologação e produção

3.2 Arquitetura Inicial

Nesta seção é apresentada a arquitetura inicial do *framework* que mantém *monitoring slices* atualizados automaticamente quando *cloud slices* são criados, modificados ou destruídos na plataforma de nuvem. Adicionalmente, o *framework* permite que administradores de nuvem construam *monitoring slices* com qualquer solução de monitoramento disponível e que melhor atenda suas necessidades. Portanto, administradores de nuvem não precisam detectar manualmente as operações realizadas sobre *cloud slices* e configurar manualmente os respectivos *monitoring slices* ou disparar *scripts* que executem estas tarefas. O *framework* é baseado em uma arquitetura modular ilustrada na Figura 3.4, que é composta por três componentes principais: *Gatherers*, *Framework Core* e *Configurators*.

Gatherers são responsáveis por coletar informações das plataformas de nuvem (Figura 3.4, passo 1) e por enviar estas informações para o *Framework Core* através do *Receptor* (Figura 3.4, passo 2). *Gatherers* podem ser desenvolvidos para coletar informações de plataformas de nuvem utilizando a interface de programação utilizada por aquela plataforma. Por exemplo, um *Gatherer* pode ser desenvolvido para coletar informações de uma plataforma utilizando uma interface como a Amazon EC2. A Amazon EC2 é a interface padrão *de facto* para o gerenciamento de computação em nuvem e é adotada por plataformas de nuvem como o OpenStack. Entretanto, também existem propostas de interfaces padrão para o gerenciamento de computação em nuvem, tais como a OGF OCCI ou a DMTF CIMI. Para o *framework*, as interfaces padrão trazem a vantagem de eliminar a necessidade do desenvolvimento de um grande número de *Gatherers* que seriam específicos para uma plataforma de nuvem. Ou seja, se *Gatherers* forem desenvolvidos para cada interface padrão, todas as plataformas de nuvem compatíveis com aquela interface padrão poderiam comunicar-se com o *framework*.

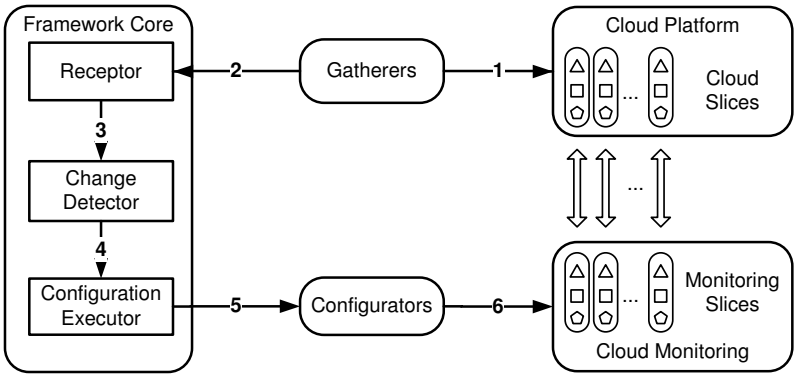


Figura 3.4: Principais componentes da arquitetura inicial: *Gatherers*, *Framework Core* e *Configurators*

Os *Gatherers* podem ser executados através de um processo de *polling*, onde serão executados uma vez a cada intervalo de tempo estabelecido pelo administrador. Outra possibilidade é o *Gatherer* ser executado exatamente após uma operação sobre um *cloud slice* na plataforma. Embora esta alternativa seja mais eficiente, por não perder tempo entre cada execução do processo de *polling*, ela depende da capacidade da plataforma de ser instruída a executar o *Gatherer* após uma operação sobre um *cloud slice*. Uma alternativa intermediária seria acompanhar algum *log* de execução da plataforma para detectar as operações sobre *cloud slices* e depois disparar o *Gatherer*. Estas últimas alternativas são dependentes de características das plataformas e podem ser exploradas por *Gatherers* especialmente projetados para estas plataformas.

O *Framework Core* é responsável por processar as informações das plataformas de nuvem recebidas dos *Gatherers*, por armazenar estas informações para permitir a detecção de mudanças (*e.g.* criação de *cloud slices*) e por disparar *Configurators* que vão construir os *monitoring slices* correspondentes aos novos *cloud slices* detectados. O *Framework Core* recebe informações dos *Gatherers* através do *Receptor*.

Depois de receber dos *Gatherers* as informações sobre as plataformas, o *Framework Core* realiza a detecção de mudanças (Figura 3.4, passo 3) na configuração atual da plataforma de nuvem (*e.g.* a criação de novos *cloud slices*). O módulo *Change Detector* verifica o identificador de cada *cloud slice* informado pelo *Gatherer* para comparar com os identificadores daqueles *cloud slices* armazenados atualmente na base de dados. Se o módulo *Change Detector* encontra um novo *cloud slice*, o módulo armazena o novo *cloud slice* na base de dados e insere a mudança detectada (criação de um novo *cloud slice*) em uma base de dados de mudanças. Obviamente, se a plataforma de nuvem possui alguma funcionalidade que é capaz de informar sobre a criação de *cloud slices*, o *framework* não precisaria de um módulo para detecção de mudanças. Entretanto, esta funcionalidade não está disponível em todas plataformas de nuvem o que exige a utilização deste módulo.

Configurators são responsáveis por receber informações do *Framework Core* e por configurar as soluções de monitoramento para coletar as métricas que formam os *monitoring slices*. *Configurators* são *scripts* que são registrados no *framework* para serem disparados para configurar as soluções de monitoramento. Portanto, as soluções de monitoramento devem ser capazes de serem configuradas por *scripts* para serem suportadas pelo *framework*. O módulo *Configuration Executor* dispara os *Configurators* e passa, como argumentos, as informações que eles precisam para configurar as soluções de monitoramento (Figura 3.4, passo 5). Dentre estas informações pode estar, por exemplo, a frequência em que uma métrica deve ser monitorada. Cada *Configurator* trata as peculiaridades das soluções de monitoramento, ou seja, geram arquivos de configuração (Nagios) ou executam *scripts* de configuração (MRTG). De fato, os *scripts* desenvolvidos pelos administradores para automatizar as tarefas de configuração podem ser registrados no *framework* como *Configurators*. Estes *scripts* podem ser beneficiados com a utilização do *framework* que recupera as informações que os *scripts* precisam e automatiza a sua execução quando um novo *cloud slice* é criado.

Tabela 3.1: Interesses de *Configurators* que são suportados pelo *framework*

Objeto	**Operações**		
	New	**Update**	**Delete**
Platform	new platform	update platform	delete platform
Cloud	new cloud	update cloud	delete cloud
Slice	new slice	update slice	delete slice
Resource	new resource	update resource	delete resource

Cada *Configurator* tem um interesse e um conjunto de condições que são verificadas antes de sua execução. Os interesses indicam os tipos de operações que são suportadas pelos *Configurators*. A Tabela 3.1 apresenta todos os interesses suportados pelo *framework*. Para cada tipo de objeto (*e.g. Platform*, *Cloud*, *Slice*, *Resource*) há 3 tipos de operações possíveis (*e.g. New*, *Update*, *Delete*). A operação *New* ocorre quando o *framework* detecta que um novo objeto foi criado na plataforma. A operação *Update* ocorre quando houver

alteração em qualquer informação relacionada a um objeto, exceto seu identificador. Já a operação *Delete* ocorre quando um objeto for apagado na plataforma de nuvem.

As condições são testes que os objetos (*e.g. Slice*, *Resource*) devem satisfazer para serem configurados pelo *Configurator*. Por exemplo, o *Configurator* `configure_mrtg`, apresentado na Figura 3.5, está interessado em mudanças de novos recursos (*new resource*) porque ele configura a solução MRTG para monitorar interfaces de rede, que podem ser representadas como recursos no *framework*. Entretanto, todo tipo de recurso de um *cloud slice* (*e.g.* CPU, memória, armazenamento) coincide neste interesse. Portanto, para restringir a execução do *Configurator* para os recursos apropriados, o *Configurator* `configure_mrtg` tem uma condição que o recurso deve satisfazer para que o *Configurator* seja executado. No caso, o identificador do recurso deve ser "*network*". Portanto, interesses e condições garantem que o *Configurator* `configure_mrtg` seja executado apenas quando um novo recurso "*network*" é criado em um *cloud slice*.

```
Nome:           configure_mrtg
Interesse:      New resource
Condição:       @resource.identifier =~ /network/
Comando:        /usr/sbin/configure_mrtg.pl
Args:           --slice_name @slice.identifier
                --ip @slice.ip
                --interface_name @resource.interface
```

Figura 3.5: Exemplos de atributos de um *Configurator*

O módulo *Configuration Executor* verifica a lista dos *Configurators* registrados no *framework*, e para cada *Configurator* o módulo busca na base de dados aquelas mudanças que são de interesse daquele *Configurator* (Figura 3.4, passo 4). Para cada mudança que é de interesse de um *Configurator*, o módulo verifica se a mudança não foi configurada em execuções anteriores por aquele *Configurator*. Se a mudança não foi configurada anteriormente, o módulo avalia se a mudança satisfaz todas as condições definidas pelo *Configurator*. Se a mudança satisfaz todas as condições, o módulo pode disparar o *Configurator* para executar a configuração da solução de monitoramento (Figura 3.4, passo 5). Entretanto, antes de disparar o *Configurator*, o módulo deve recuperar os argumentos necessários pelo *Configurator*. Os argumentos definem como o *Framework Core* comunica-se com os *Configurators* e são assinalados pelo "`@`" na definição do *Configurator*. Por exemplo, para o *Configurator* `configure_mrtg` ilustrado na Figura 3.5, o módulo precisa recuperar do modelo de informação os argumentos `@slice.identifier`, `@slice.ip` e `@resource.interface`. Depois de recuperar os argumentos, o módulo dispara o *Configurator* utilizando os argumentos e armazena sua saída para análise futura ou verificação de problemas.

Além de determinar as mudanças que serão avaliadas, o interesse de um *Configurator* determina o escopo em que o modelo de informação será consultado. Por exemplo, caso o interesse de um *Configurator* seja por *new slice*, o modelo de informação poderá selecionar atributos do nível *Slice* e de seus respectivos objetos dos níveis superiores: *Cloud* e *Platform*, conforme exemplo de instância do modelo ilustrado na Figura 3.3. Logo, qualquer atributo destes níveis poderá ser utilizado como argumento pelo *Configurator*. Estes atributos são acessados diretamente através dos seletores `@resource.atributo`, `@slice.atributo`, `@cloud.atributo` e `@platform.atributo`, para atributos dos níveis *Resource*, *Slice*, *Cloud* e *Platform*, respectivamente.

A arquitetura inicial do *framework* possibilita que *monitoring slices* sejam configurados independentemente das soluções de monitoramento e plataformas de nuvem empregadas no ambiente de computação em nuvem. Entretanto, na arquitetura inicial a requisição gerada quando um *Gatherer* envia informações para o *Framework Core* e todas as etapas referentes ao processamento desta requisição são executadas como uma grande tarefa monolítica. Ou seja, a requisição é processada por apenas uma *thread* do *framework*. Durante a etapa de avaliação da arquitetura inicial, esta característica prejudicou o tempo de resposta do *framework* na configuração de *monitoring slices*. Como consequência, a arquitetura inicial não alcança a escalabilidade necessária e não é apropriada para o monitoramento de ambientes computacionais em nuvem.

As soluções de monitoramento são instaladas em servidores de monitoramento. A quantidade de métricas a serem monitoradas e de soluções de monitoramento utilizadas pode exigir a utilização de vários servidores. Neste aspecto, surge outra desvantagem da arquitetura inicial do *framework* em relação ao processo de definição do servidor que será configurado para monitorar as métricas de *monitoring slices* de um determinado tipo. Este processo é denominado atribuição de tarefas de configuração. Na arquitetura inicial, os administradores da nuvem devem atribuir manualmente as métricas que devem ser monitoradas por cada servidor de monitoramento, *e.g.* métricas sobre a CPU de *monitoring slices* do tipo #1 serão monitoradas pelo servidor Nagios #A, já métricas de CPU de *monitoring slices* do tipo #2 serão monitoradas pelo servidor Nagios #B. Portanto, a atribuição de tarefas de configuração é realizada estaticamente.

Além destas desvantagens, a arquitetura inicial não leva em conta que a carga dos servidores de monitoramento varia ao longo do tempo. Portanto, a atribuição manual e estática de tarefas de configuração pode levar a uma distribuição desbalanceada das tarefas de configuração e consequentemente desbalancear a carga gerada pelo monitoramento entre os servidores de monitoramento. Por sua vez, esta estratégia pode levar a potenciais perdas de desempenho no monitoramento da nuvem.

Nesta seção, os principais componentes da arquitetura inicial foram apresentados. Além disso, foram discutidas algumas desvantagens desta arquitetura que levaram à necessidade de melhoria do desempenho e inclusão de novas funcionalidades ao *framework*. Para melhorar o desempenho da solução, foi necessário modificar a arquitetura inicial ao introduzir novos componentes. Os novos componentes e funcionalidades introduzidos na arquitetura estendida são apresentados na próxima seção.

3.3 Arquitetura Estendida

Nesta seção é apresentada a arquitetura estendida que foi projetada para dividir o processamento de uma requisição em diversas tarefas menores que pudessem ser processadas paralelamente. Neste sentido, a arquitetura estendida utiliza filas e trabalhadores onde tarefas são colocadas em filas e processadas por trabalhadores, sendo que pode-se ter diversos trabalhadores processando simultaneamente tarefas do mesmo tipo. Desta forma, o *framework* é capaz de explorar melhor o paralelismo disponível atualmente nas arquiteturas dos computadores e melhorar o desempenho do *framework*.

Além das melhorias no desempenho introduzidas pela utilização de paralelismo, a arquitetura estendida possibilita a introdução de duas novas funcionalidades: atribuição dinâmica e automática de tarefas de configuração para servidores de monitoramento; e balanceamento de carga entre servidores de monitoramento durante a configuração de *monitoring slices*, que são detalhadas a seguir.

A primeira funcionalidade é a atribuição dinâmica e automática de tarefas de configuração. Esta funcionalidade possibilita que métricas de *monitoring slices* de um tipo sejam atribuídas a grupos de servidores de monitoramento. Isto significa que não há um servidor de monitoramento específico e atribuído estaticamente que será configurado para monitorar métricas de um *monitoring slice*. Na arquitetura estendida, qualquer servidor de monitoramento pode registrar-se em um grupo para receber configurações de métricas de *monitoring slices* atribuídas automaticamente para aquele grupo.

A segunda funcionalidade é o balanceamento de carga durante a configuração dos *monitoring slices*. Esta funcionalidade possibilita que os servidores de monitoramento possam se voluntariar para receber tarefas de configuração quando possuírem capacidade para receber novas tarefas; ou parar de receber novas tarefas enquanto não possuírem capacidade. A decisão de se voluntariar para receber novas tarefas pode ser baseada em diversos critérios. No protótipo apresentado no Capítulo 4, o critério utilizado é a carga atual de processamento no servidor, cujo limite aceitável é informado pelo administrador.

A arquitetura estendida do *framework* tem mudanças estruturais significativas no *Framework Core* para melhorar o desempenho e introduzir as duas funcionalidades no *framework*. Esta nova arquitetura é ilustrada na Figura 3.6, onde os componentes em cinza indicam as mudanças introduzidas. Nesta seção, são apresentadas as filas e trabalhadores introduzidos para dividir o processamento de requisições em tarefas menores.

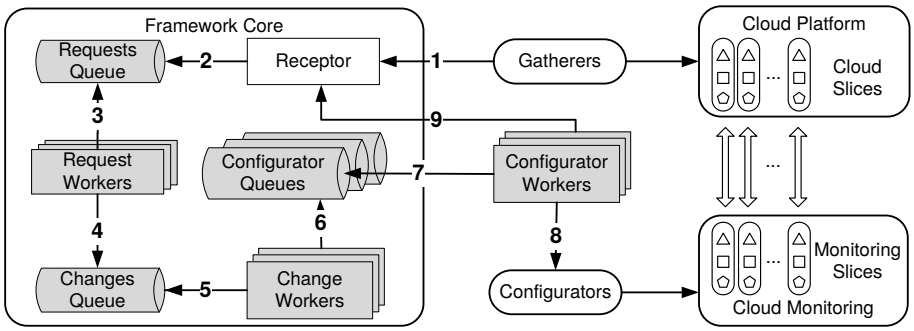


Figura 3.6: Arquitetura estendida através de filas e trabalhadores

O *Framework Core* recebe dos *Gatherers* informações sobre os *cloud slices* através do *Receptor* (Figura 3.6, passo 1). O *Framework Core* realiza o processamento das informações recebidas até disparar os *Configurators* apropriados (Figura 3.6, passos 2-6). Este processamento é realizado por filas e trabalhadores para dividir o processamento da requisição em tarefas menores. Primeiramente, a informação recebida é enfileirada na *Requests Queue* (Figura 3.6, passo 2) para ser processada por um *Request Worker* (Figura 3.6, passo 3). *Request Workers* são responsáveis por detectar mudanças entre as informações recebidas e as informações armazenadas sobre os *cloud slices* hospedados na plataforma de nuvem. Cada mudança detectada é enfileirada na *Changes Queue* (Figura 3.6, passo 4) para ser processada por um *Change Worker* (Figura 3.6, passo 5). *Change Workers* são responsáveis por avaliar se a mudança detectada satisfaz o interesse e as condições dos *Configurators*, que são regras predefinidas pelo administrador da nuvem. Por exemplo, estas regras são verificadas para determinar se uma mudança referente a um novo *cloud slice* exige a criação de um novo *monitoring slice* correspondente. Quando todas estas regras são satisfeitas, o *Change Worker* enfileira uma *configurator call* na *Configurator Queue* (Figura 3.6, passo 6) indicada pelo administrador da nuvem. A *configurator call* armazena uma identificação e o comando que executa o *Configurator* com todos os argumentos necessários para configurar a solução de monitoramento.

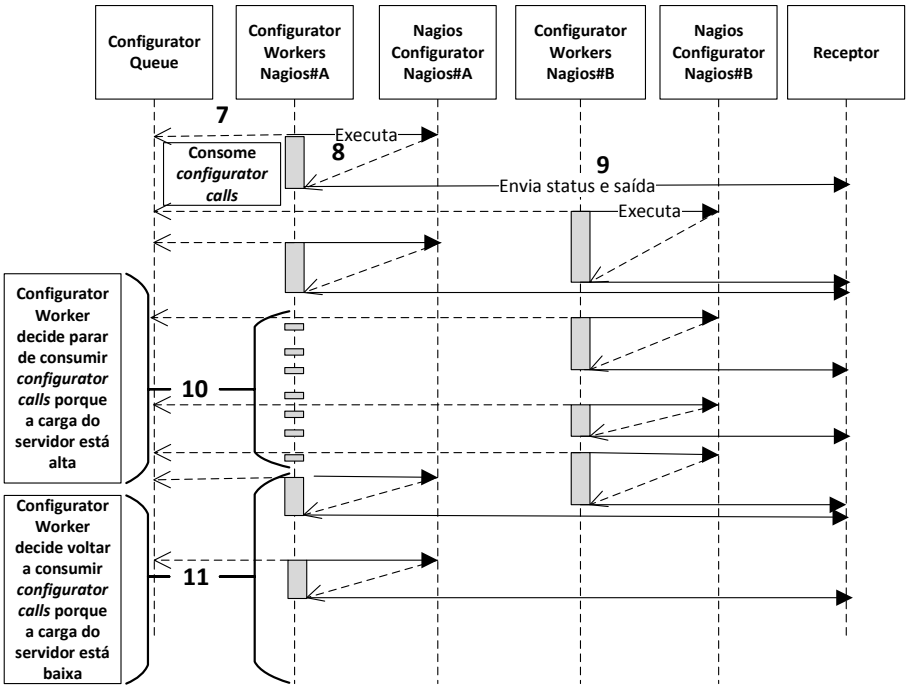


Figura 3.7: Diagrama de sequência sobre as trocas de mensagens entre *Configurator Workers*, *Configurator Queues*, *Configurators* e *Receptor*

A Figura 3.7 mostra as iterações 7-11 da Figura 3.6 do ponto de vista da concorrência entre *Configurator Workers* executados em servidores de monitoramentos distintos. Desta forma, *Configurator Workers* são responsáveis por consumir as *configurator calls* das *Configurator Queues* apropriadas (Figuras 3.6 e 3.7, passo 7). O *Configurator Worker* executa o *Configurator* (Figuras 3.6 e 3.7, passo 8) e armazena o código de *status* da execução e a sua saída para enviar para o *Framework Core* através do *Receptor* (Figuras 3.6 e 3.7, passo 9). Estas informações são utilizadas pelo administrador da nuvem para análise futura e detecção de problemas. *Configurator Workers* levam em consideração a carga do servidor de monitoramento. Portanto, *Configurator Workers* podem decidir parar de consumir *configurator calls* enquanto a carga do servidor estiver inaceitável (Figura 3.7, período 10). Quando a carga do servidor retorna para um valor aceitável, o *Configurator Worker* decide consumir *configurator calls* novamente (Figura 3.7, período 11).

Configurator Queues representam grupos de servidores de monitoramento que compartilham características atribuídas pelo administrador da nuvem. Estas características podem estar relacionadas às soluções de monitoramento que estão disponíveis no servidor de monitoramento, ou à determinada capacidade atribuída pelo administrador àquele servidor. A Tabela 3.2 apresenta exemplos de características atribuídas aos servidores de monitoramento. No exemplo da Tabela 3.2, as características são as soluções de monitoramento disponíveis e o tipo de assinatura MaaS dos *cloud slices* que serão monitorados. Variando-se estas 2 características, tem-se 4 grupos de servidores e consequentemente 4 filas: nagios_basic, mrtg_basic, nagios_platinum e mrtg_platinum.

Cada *Configurator* é associado a uma fila (*Configurator Queue*) onde suas *configurator calls* serão enfileiradas. Como *Configurators* são disparados apenas quando seu interesse e suas condições são satisfeitos, pode-se utilizar interesses e condições para determinar em que fila as *configurator calls* serão enfileiradas. Consequentemente, também será determinado o grupo de servidores ao qual a *configurator call* será atribuída. Ou

seja, com base em informações dos *cloud slices* é escolhido o grupo de servidores que irá coletar as métricas dos seus *monitoring slices*.

Tabela 3.2: Exemplos de *Configurator Queues*

Soluções de Monitoramento	**Assinatura MaaS**	
	Basic	**Platinum**
Nagios	nagios_basic	nagios_platinum
MRTG	mrtg_basic	mrtg_platinum

A Tabela 3.3 apresenta exemplos de interesses e condições para atribuir tarefas de configuração para grupos de servidores conforme as filas da Tabela 3.2. Neste exemplo, as *Configurator Queues* são selecionadas de acordo com a solução de monitoramento que será utilizada e a assinatura MaaS adquirida pelo usuário para o *cloud slice*. Para cada *cloud slice*, três métricas serão monitoradas: o estado do *host* (ping) e a utilização de CPU pelo Nagios e de rede pelo MRTG. As tarefas de configuração para o Nagios são atribuídas para as filas nagios_basic e nagios_platinum de acordo com a assinatura MaaS que é selecionada de acordo com o atributo `@slice.MaaS`. Este atributo é recuperado diretamente do modelo de informação e indica o tipo de assinatura MaaS adquirida para aquele *cloud slice* (*basic* ou *platinum*). O mesmo ocorre para as tarefas de configuração para o MRTG que são atribuídas para as filas mrtg_basic e mrtg_platinum.

Tabela 3.3: Exemplos de atributos de *Configurators*

Configurator	**Atributo**	**Valor**
Nome: nagios_host_basic	Interesse	New Slice
Fila: nagios_basic	Condição	@slice.MaaS =~ /basic/
Nome: nagios_cpu_basic	Interesse	New Resource
Fila: nagios_basic	Condição	@resource.identifier =~ /CPU/
	Condição	@slice.MaaS =~ /basic/
Nome: mrtg_basic	Interesse	New Resource
Fila: mrtg_basic	Condição	@resource.identifier =~ /network/
	Condição	@slice.MaaS =~ /basic/
Nome: nagios_host_plat	Interesse	New Slice
Fila: nagios_platinum	Condição	@slice.MaaS =~ /platinum/
Nome: nagios_cpu_plat	Interesse	New Resource
Fila: nagios_platinum	Condição	@resource.identifier =~ /CPU/
	Condição	@slice.MaaS =~ /platinum/
Nome: mrtg_plat	Interesse	New Resource
Fila: mrtg_platinum	Condição	@resource.identifier =~ /network/
	Condição	@slice.MaaS =~ /platinum/

Configurator Queues e *Configurator Workers* possibilitam a introdução das duas novas funcionalidades introduzidas no *framework*. A atribuição dinâmica e automática de tarefas de configuração é alcançada quando as *configurator calls* são atribuídas a grupos de servidores de monitoramento (*Configurator Queues*) que são selecionados de acordo

com regras predefinidas pelo administrador. Esta abordagem difere da arquitetura inicial, onde as tarefas de configuração eram atribuídas estaticamente pelo administrador para um servidor, o que dificulta a distribuição de tarefas entre vários servidores de monitoramento. O balanceamento de carga é alcançado quando *Configurator Workers* consomem concorrentemente uma mesma *Configurator Queue*. A Figura 3.7 apresenta a concorrência por *configurator calls* de dois servidores de monitoramento Nagios (Nagios#A e Nagios#B). Esta abordagem também possibilita que servidores de monitoramento decidam parar de consumir tarefas de configuração baseados na sua carga (Figura 3.7, período 10), pois *Configurator Workers* levam em consideração a carga do servidor. Da mesma forma, *Configurator Workers* podem decidir retornar a consumir tarefas de configuração quando a carga do servidor retornar para um nível aceitável (Figura 3.7, período 11).

Este capítulo apresentou as arquiteturas inicial e estendida do FlexACMS e discutiu a evolução da solução, para dessa forma, justificar decisões tomadas nas etapas de projeto e implementação do *framework*. Embora a arquitetura inicial seja capaz de atender os objetivos propostos para esta dissertação, a arquitetura estendida deve ser utilizada em virtude dos ganhos de desempenho, que são apresentados no Capítulo 5, e das duas funcionalidades introduzidas apresentadas nesta seção.

4 IMPLEMENTAÇÃO DO PROTÓTIPO

Neste capítulo são apresentados os detalhes de implementação dos componentes da arquitetura estendida do FlexACMS ilustrada na Figura 3.6. Conforme discutido no final do Capítulo 3, esta arquitetura é sugerida como solução para a construção automatizada de *monitoring slices*. Na Seção 4.1 é apresentada a implementação do *Framework Core* desenvolvido para a arquitetura estendida. Na Seção 4.2 é detalhada a implementação do *Gatherer* desenvolvido para a plataforma OpenStack utilizando a OpenStack API. Na Seção 4.3 é apresentada a implementação do *Configurator Worker* que leva em consideração a carga do servidor de monitoramento. Na Seção 4.4 são apresentados os detalhes de implementação dos *Configurators* desenvolvidos para o Nagios e MRTG.

4.1 *Framework Core*

O *Framework Core* é o componente principal do FlexACMS, pois agrupa as principais funcionalidades do *framework*, como detecção de mudanças e o processamento de interesses e condições dos *Configurators*. Além disso, é o componente central da solução e comunica-se com *Gatherers* e *Configurators*.

A comunicação entre o *Framework Core* e os *Gatherers*, apesar de pertencerem a mesma solução, é semelhante à comunicação entre sistemas distintos. Portanto, deve-se adotar um padrão na troca de mensagens entre estes componentes para permitir que administradores e desenvolvedores possam desenvolver *Gatherers*. Neste contexto, a utilização de *Service Oriented Architecture* (SOA) (ERL, 2008) e de *Web services* é apropriada por permitir a definição de um padrão de comunicação através de uma Interface de Programação de Aplicações (API - *Application Programming Interface*). Dentre as tecnologias para a implementação de *Web services*, o REST adota um padrão mais simples para a troca de mensagens, onde os tipos de mensagens são baseados nos verbos do protocolo *HyperText Transfer Protocol* (HTTP) (*e.g.* GET, PUT, POST) (RICHARDSON; RUBY, 2008). Esta simplicidade auxilia tanto no desenvolvimento dos *Gatherers* como dos *Web services*, e por isso o FlexACMS adota o REST como padrão para seus *Web services*.

O *Framework Core* foi desenvolvido utilizando o *framework* de desenvolvimento *Ruby on Rails* (HARTL; FERNANDEZ, 2011), que foi escolhido pela facilidade de desenvolvimento de interfaces de usuário baseadas na *Web* e REST *Web services*. O *Ruby on Rails* utiliza o *design pattern*[1] *Model-View-Controller* (MVC) para a arquitetura das aplicações desenvolvidas no *framework*. O *design pattern* MVC é um modelo de aplicação baseado em três papéis: *Model*, *View* e *Controller*. O *Model* contém os dados e

[1]Embora o termo possa ser traduzido como "padrão de projeto", preferiu-se utilizar o termo em inglês pela sua utilização entre os desenvolvedores

determina os relacionamentos entre as classes. Os *Controllers* tratam as entradas fornecidas pelos usuários. As *Views* são utilizadas para mostrar informações para os usuários (BUSCHMANN; HENNEY; SCHMIDT, 2007).

A facilidade em desenvolver interfaces *Web* e REST *Web services* no *framework Ruby on Rails* advém da automação introduzida nas interações entre os componentes responsáveis pelos papéis do *design pattern* MVC. Entretanto, o desenvolvedor deve adotar as convenções estabelecidas pelo *framework* para usufruir desta automação, pois o *Ruby on Rails* adota o paradigma *Convention over Configuration (CoC)* (HARTL; FERNANDEZ, 2011). CoC preconiza a utilização de configurações implícitas que são chamadas de convenções, que são automaticamente instanciadas pela aplicação. Algumas destas convenções são apresentadas no decorrer desta seção.

A Figura 4.1 ilustra as tecnologias e os componentes utilizados no desenvolvimento e implantação do *Framework Core*. Para instanciar o *Framework Core* foi utilizado o módulo *Passenger* para o servidor *Web* Apache (Figura 4.1, passo 1). O *Passenger* é bastante utilizado por permitir que aplicações *Ruby on Rails* sejam fornecidas pelo servidor *Web* Apache. O banco de dados *MySQL* foi escolhido para persistir os dados dos *Models* (passo 2). Para armazenar as informações sobre filas e trabalhadores foi utilizado o Redis (passo 3), que é uma base de dados para chaves e valores (*Key-value Database*). Para manipular tarefas nas filas armazenadas no Redis foi utilizada uma biblioteca para *Ruby on Rails* chamada Resque (HARTL; FERNANDEZ, 2011) (passos 3 e 4). Através da biblioteca Resque, também pode-se desenvolver trabalhadores na linguagem Ruby, ou seja, na mesma linguagem utilizada para o desenvolvimento dos *Controllers*.

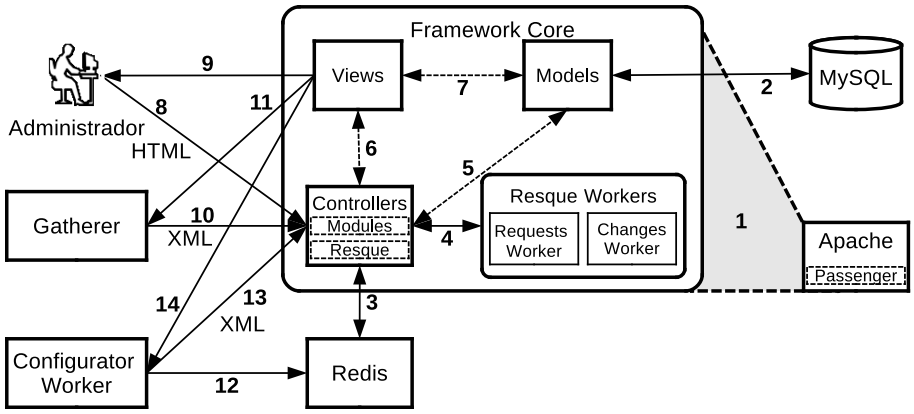


Figura 4.1: Implementação de componentes do *Framework Core*

Os relacionamentos de *Controllers* com *Models* e *Views* (passos 5 e 6, respectivamente), de *Views* com *Models* (passo 7), o funcionamento da interface *Web* utilizada pelos administradores (passos 8 e 9) e do REST *Web service* (passos 10, 11, 13 e 14) são apresentados durante esta seção. Nas Seções 4.2 e 4.3 são apresentadas as interações dos *Gatherers* e *Configurator Worker* com o REST *Web service* fornecido pelo *Framework Core* (passos 10, 11, 13 e 14). Na Seção 4.3 também são apresentadas as interações do *Configurator Worker* com o Redis (passo 12).

A Figura 4.2 apresenta o código-fonte dos métodos `index` e `create` do *Controller* `PlatformsController`. O método `index` é responsável por listar todas as plataformas armazenadas. O método `create` é responsável por criar o objeto referente a uma plataforma e é invocado toda vez que um *Gatherer* envia informações para o *framework*. Os *Controllers* relacionam-se com outros componentes dos papéis *View* e *Model* do *design pattern* MVC. O relacionamento com o *Model* está codificado nas linhas 3, 12 e 15 da

Figura 4.2 através dos métodos `all`, `new` e `save` do *Model* `Platform`, que recuperam todas as plataformas, instanciam um objeto para uma nova plataforma e persistem as informações do objeto no banco de dados, respectivamente. O relacionamento com as *Views* está codificado nas linhas 6,7,16,17,19 e 20 da Figura 4.2 através da instrução `format` que, por convenção, utilizará o *template* da *View* associado ao método e formato escolhido. Para o método `index` podem ser utilizados os *templates* `index.xml.builder` ou `index.html.erb` dependendo do formato escolhido (*eXtensible Markup Language* (XML) ou *HyperText Markup Language* (HTML)) como mostram os comentários das linhas 6 e 7. Na linha 24 está codificado o relacionamento com a biblioteca Resque utilizada pelo *Controller* para enfileirar o `id` da nova plataforma criada a partir de uma requisição de um *Gatherer*. As informações associadas a esta nova plataforma serão processadas pelo *Module* `ChangeDetection` que é apresentado no decorrer desta seção.

```
1  class PlatformsController < ApplicationController
2    def index
3      @platforms = Platform.all
4
5      respond_to do |format|
6        format.xml  # index.xml.builder
7        format.html # index.html.erb
8      end
9    end
10
11   def create
12     @platform = Platform.new(params[:platform])
13
14     respond_to do |format|
15      if @platform.save
16         format.xml {redirect_to @platform}
17         format.html { redirect_to @platform, notice: 'Platform was
               successfully created.' }
18       else
19         format.xml
20         format.html { render action: "new" }
21       end
22     end
23
24     Resque.enqueue(ChangeDetection, @platform.id)
25   end
26 end
```

Figura 4.2: Código-fonte parcial do *Controller* `PlatformsController`

No *Ruby on Rails*, as convenções determinam o roteamento de requisições para os métodos das classes apropriadas para o processamento daquela requisição. Por convenção, uma requisição com o verbo HTTP `GET` para o nome de uma classe no plural será roteada para o método `index` do *Controller* daquela classe, cujo nome também é determinado por convenção ao concatenar `"Controller"` ao nome da classe no plural. Outra convenção diz que uma requisição com o verbo HTTP `PUT` para o nome de uma classe no singular será roteada para o método `create` do *Controller* daquela classe. Por exemplo, os métodos `index` e `create` da classe `PlatformsController`, cujos códigos estão na Figura 4.2, são invocados quando recebe-se requisições `"GET /platforms"` e `"PUT /platform"`, respectivamente. A requisição `"PUT /platform"` deve estar

acompanhada das informações que são necessárias para a instanciação do novo objeto do *Model* `Platform`, conforme codificado na linha 12 do código-fonte da Figura 4.2 que recupera estas informações através da instrução `params`.

```
1 class Platform < ActiveRecord::Base
2   attr_accessible :identifier
3   attr_protected :id
4   has_many :platform_information, :dependent => :destroy
5   has_many :cloud, :dependent => :destroy
6   accepts_nested_attributes_for :platform_information, :cloud
7 end
```

Figura 4.3: Código-fonte do *Model* para a classe `Platform`

A Figura 4.3 apresenta o código-fonte para o *Model* da classe `Platform` desenvolvido de acordo com o modelo de informação do FlexACMS. O *Model* determina quais atributos são acessíveis (`identifier`), quais atributos são protegidos (`id`) e os relacionamentos com outras classes e suas respectivas cardinalidades (`has_many :platform_information` e `has_many :cloud`). Os métodos `all`, `new` e `save` utilizados nos *Controllers* são herdados da classe `ActiveRecord::Base`. Esta classe implementa estes métodos, entre outros, de maneira que a aplicação seja independente do banco de dados utilizado. No *Framework Core*, foi utilizado o MySQL para persistir as informações, cujas instruções *Structured Query Language* (SQL) são geradas em tempo de execução pela classe `ActiveRecord::Base`.

As requisições recebidas são roteadas, por convenção, para serem processadas por um mesmo método independentemente do formato da requisição (*e.g.* HTML, XML, *JavaScript Object Notation* (JSON)). Após o processamento, o método responde à requisição utilizando o mesmo formato recebido na requisição, conforme codificado nas linhas 5-8 do código-fonte da Figura 4.2. A chamada `format.<formato>` repassa as informações recuperadas pelo *Model* para a *View* associada àquele método. Esta *View* possui um *template* associado para construir a resposta de acordo com o formato da requisição.

A Figura 4.4 apresenta o *template* do formato XML para o método `index` da classe `Platform`. Este *template* utiliza métodos dos *Models* para acessar as informações recuperadas e gerar a saída que será enviada pelas *Views*. No exemplo, as informações enviadas pelo *Controller* estão armazenadas no vetor `@platforms`. Este vetor é percorrido pelo método `each` que associa o conteúdo de um item do vetor a uma variável (linha 4). Na linha 5, é gerado na resposta um nodo XML `<platform>`. Já na linha 6, é gerado na resposta um nodo XML `<identifier>` que receberá o conteúdo do atributo `identifier` da variável `platform`. Através dos *templates* também pode-se percorrer os relacionamentos de uma classe, conforme mostram as linhas 8 e 16 que percorrem os relacionamentos de `platform` com `platform_information` e `cloud`, respectivamente. Desta forma, o XML de resposta do REST *Web service* é gerado recursivamente. Para interfaces *Web*, utilizam-se métodos semelhantes, mas que geram a saída em HTML.

A biblioteca Resque é utilizada para manipular filas e trabalhadores, cujas informações são armazenadas no Redis. As tarefas são enfileiradas pelos *Controllers* para serem processadas pelos trabalhadores. Conforme exigido pela biblioteca Resque, os trabalhadores executam o código de um *Module* que define um novo espaço de nomes dentro da aplicação. Cada *Module* é associado a uma fila do Resque. Por convenção, os trabalhadores executam o método `perform` do *Module* associado à fila.

```
1  xml.instruct! :xml, :version=>"1.0"
2
3  xml.platforms(:type => 'array') do
4    @platforms.each do |platform|
5      xml.platform do
6          xml.identifier platform.identifier
7          xml.platform_information_attributes(:type => 'array') do
8           platform.platform_information.each do |platform_information|
9            xml.platform_information do
10            xml.name  platform_information.name
11            xml.value  platform_information.value
12           end
13          end
14         end
15         xml.cloud_attributes(:type => 'array') do
16          platform.cloud.each do |cloud|
17           xml.cloud do
18            xml.identifier cloud.identifier
19           end
20          end
21         end
22     end
23   end
24 end
```

Figura 4.4: Código-fonte parcial do *template* XML para o método `index` da classe `PlatformsController`

A Figura 4.5 apresenta o código-fonte do *Module* `ChangeDetection` que é responsável pela detecção de mudanças. Este *Module* está associado à fila `platform_check`, conforme mostra a linha 3 da Figura 4.5. A detecção de mudanças é iniciada na linha 6 da Figura 4.5 e utiliza duas instâncias do modelo de informação descrito na Seção 3.1, que são criadas na base de dados do *Framework Core*. As informações enviadas pelos *Gatherers* são armazenadas na instância *Pending Information* e as informações consolidadas no *framework* são armazenadas na instância *Processed Information*. O processo de detecção de mudanças compara as informações da instância *Pending Information* com as informações da instância *Processed Information*. Caso encontre uma diferença entre as informações armazenadas nestas instâncias, o *Module* `ChangeDetection` atualiza as informações da instância *Processed Information*, adiciona uma mudança na tabela de mudanças e enfileira uma nova tarefa para a verificação desta mudança que será processada por um outro *Module* chamado `ChangeTasks`.

```
1  module ChangeDetection
2
3    @queue = :platform_check
4
5    def self.perform(platform_id)
6      ChangeDetection.check_platforms(platform_id)
7    end
8  end
```

Figura 4.5: Código-fonte parcial do *Module* `ChangeDetection`

A verificação da mudança é realizada por outro *Module* chamado `ChangeTasks` associado à fila `change_check`. Este *Module* é responsável por decidir se é necessário executar um *Configurator* avaliando interesses, condições e o *status* de execuções anteriores deste *Configurator* para a mesma mudança. Para o *framework*, uma execução de *Configurator* tem sucesso se terminar com *status* "0". O módulo concatena o comando do *Configurator* com seus argumentos, substituindo os atributos indicados como `@objeto.atributo` pelas informações do modelo de informação utilizando o escopo definido pelo interesse do *Configurator*. Desta forma, será gerado o comando da *configurator call*, que é armazenada em uma tabela de execuções de *Configurators* e recebe um identificador. Este identificador e seu comando são enfileirados na *Configurator Queue* associada ao *Configurator* para evitar que o *Configurator Worker* tenha que realizar uma consulta adicional ao *Framework Core* para recuperar o comando, caso apenas o identificador fosse enfileirado. Desta forma, o *Configurator Worker* recupera o comando e o identificador da *configurator call* diretamente da fila e, após executar o comando, entra em contato com o *Framework Core* apenas para informar o *status* da execução do comando e sua saída utilizando o identificador da *configurator call*. As *configurator calls*, seus comandos e identificadores são apresentados em detalhes na Seção 4.3.

Tabela 4.1: Número de linhas de código necessárias para codificar cada componente

Componente	**Número de linhas de código**
Models	158 linhas
Views	2991 linhas
Controllers	2743 linhas
Modules	311 linhas

O número de linhas de código é utilizado como medida para o esforço de desenvolvimento do protótipo. A Tabela 4.1 mostra o número total de linhas de código utilizadas para cada tipo de componente do *framework Ruby on Rails* para codificar o *Framework Core*. No total, o *Framework Core* possui 6223 linhas de código distribuídas pelos diferentes tipos de componentes. Deste total, quase 50% das linhas foram utilizadas para codificar os *templates* para as *Views*, que são utilizados em sua maioria para o desenvolvimento da interface *Web*. Outra parcela significativa de linhas de código é utilizada para a codificação dos *Controllers*. Estes componentes coordenam o fluxo de informações dentro da aplicação, cujo esforço de desenvolvimento é bem mais elevado do que o exigido para *Models* e *Views*. Nos *Modules* encontram-se os códigos responsáveis pela detecção de mudanças e pela avaliação de interesses e condições dos *Configurators*, cujo esforço de desenvolvimento é semelhante ao exigido no desenvolvimento de *Controllers*.

4.2 *Gatherers*

Os *Gatherers* são os componentes responsáveis por coletar informações das plataformas de computação em nuvem e encaminhá-las para o *Framework Core* através do REST *Web service*. Os *Gatherers* utilizam a API disponibilizada pela plataforma para coletar informações sobre a plataforma e sobre os *cloud slices* hospedados por ela. As informações coletadas devem ser organizadas de acordo com o formato estabelecido pelo REST *Web service* antes de serem enviadas para o *Framework Core*.

A Figura 4.6 ilustra as tecnologias e os componentes utilizados para o desenvolvimento do *Gatherer* para a OpenStack API. Este *Gatherer* foi desenvolvido utilizando a linguagem Python e utiliza uma biblioteca para comunicação com *Web services* chamada `httplib`. Esta biblioteca foi utilizada tanto para a comunicação do *Gatherer* com o OpenStack como para a comunicação do *Gatherer* com o *Framework Core*.

O *Gatherer* para a OpenStack API é executado através de um processo de *polling* controlado pelo sistema *Cron* do servidor (Figura 4.6, passo 1). As primeiras interações do *Gatherer* são direcionadas ao serviço de identidades do OpenStack chamado *Keystone*. Estas interações (Figura 4.6, passos 2 e 3) são necessárias para recuperar o *token* de autenticação para o usuário e o identificador do *tenant* associado ao usuário informado, respectivamente. Estas informações são necessárias para as demais interações do *Gatherer* com serviço de computação do OpenStack chamado *Nova*.

A primeira interação com o *Nova* (Figura 4.6, passo 4) é necessária para recuperar o *token* de autenticação e o endereço do serviço de computação associado ao *tenant* do usuário, que pode ser diferente do endereço do serviço de computação que foi utilizado para a autenticação. A próxima interação com o *Nova* (Figura 4.6, passo 5) coleta as informações relativas aos *flavors* disponibilizados pela plataforma. Um *flavor* é um *template* para os recursos virtuais de um *cloud slice*, por exemplo, o *flavor* `m1.small` identifica o *template* de um *cloud slice* com: 1 CPU virtual, 2048 MB de memória e 20 GB de armazenamento. Estas informações são utilizadas para complementar as informações sobre os recursos de cada *cloud slice*. A última interação com o *Nova* (Figura 4.6, passo 6) recupera as informações sobre todos os *cloud slices* hospedados na plataforma.

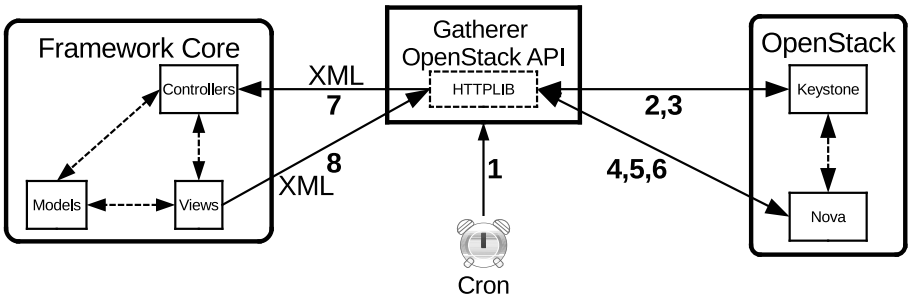


Figura 4.6: Implementação do *Gatherer* desenvolvido para a OpenStack API

De posse das informações sobre os *cloud slices* e seus *flavors*, o *Gatherer* constrói uma requisição para enviá-las para o *Framework Core* (Figura 4.6, passo 7). Esta requisição segue o formato XML estabelecido pelo REST *Web service*, conforme ilustrado na Figura 4.7. Este XML segue a estrutura de relacionamentos entre os níveis do modelo de informação do FlexACMS. Neste modelo de informação, cada nível possui um atributo chamado `identifier` ilustrado nas linhas 3, 6 e 9, que armazenam os identificadores para *Platform*, *Cloud* e *Slice*, respectivamente. As informações sobre cada um destes níveis são armazenadas em *arrays* XML. As linhas 10 a 19 mostram alguns atributos relacionados ao nível *Slice* `instance-000027c5`, como `status` (linhas 11-14) e `physical_hostname` (linhas 15-18). A mesma lógica é adotada para representar o nível *Resource* relacionado a este nível *Slice*, cujas informações estão omitidas na linha 21. O *Gatherer* envia este XML para o *Framework Core* e recebe uma resposta que informa sobre o sucesso (ou erro) no processamento da requisição (Figura 4.6, passo 8).

O *Gatherer* desenvolvido para a OpenStack API possui 450 linhas de código. As linhas de código necessárias para seu desenvolvimento dividem-se praticamente em duas

funções: construção do XML para compor a requisição do REST *Web service* e para a troca de mensagens entre o *Gatherer*, o *Keystone*, o *Nova* e o REST *Web service* utilizando objetos da classe `httplib` da linguagem Python. Entretanto, o principal esforço no desenvolvimento deste *Gatherer*, que não reflete-se na quantidade de linhas de código, foi em relação ao entendimento e integração com a OpenStack API, juntamente com o mapeamento das informações disponibilizadas por esta API para utilização no *framework*.

```
<?xml version="1.0" encoding="UTF-8"?>
<platform>
 <identifier>Platform</identifier>
  <cloud_attributes type="array">
   <cloud>
    <identifier>Cloud</identifier>
     <slices_attributes type="array">
      <slice>
       <identifier>instance-000027c5</identifier>
       <slice_information_attributes type="array">
        <slice_information>
         <name>status</name>
         <value>ACTIVE</value>
        </slice_information>
        <slice_information>
         <name>physical_hostname</name>
         <value>rsopenstack</value>
        </slice_information>
       </slice_information_attributes>
       <resource_attributes type="array">
                ...
       </resource_attributes>
      </slice>
     </slices_attributes>
   </cloud>
  </cloud_attributes>
</platform>
```

Figura 4.7: Arquivo XML gerado pelo *Gatherer* desenvolvido para a OpenStack API

4.3 *Configurator Workers*

Os *Configurator Workers* são os componentes responsáveis por recuperar as *configurator calls* das *Configurator Queues*. A *configurator call* possui o comando e os argumentos necessários para a execução dos *Configurators*. O *Configurator Worker* executa o comando com os argumentos recuperados e envia o código de *status* e saída para o *Framework Core* através do REST *Web service*. Além disso, os *Configurator Workers* levam em consideração a carga do servidor de monitoramento e podem decidir se o servidor possui capacidade para recuperar novas tarefas de configuração. Desta forma, são fundamentais para garantir que nenhum servidor de monitoramento fique sobrecarregado.

A Figura 4.8 ilustra as tecnologias e os componentes utilizados para o desenvolvimento do *Configurator Worker* que foi codificado na linguagem Perl (SCHWARTZ; PHOENIX, 2008). O *Configurator Worker* utiliza a biblioteca `Redis::Client` para comunicar-se com o Redis para recuperar as *configurator calls*. Também utiliza a biblioteca `LWP::Simple` para comunicar-se com o REST *Web service*.

O *Configurator Worker* utiliza o arquivo `/proc/loadavg`, que armazena as médias de processos em execução para os últimos 1, 5 e 10 minutos, para recuperar a carga do servidor (Figura 4.8, passo 1). O *Configurator Worker* utiliza a média de processos em execução no último minuto para determinar se o servidor possui capacidade para recuperar novas tarefas de configuração. Um valor máximo aceitável deve ser informado pelo administrador, pois este valor depende da quantidade de processos que cada servidor pode executar simultaneamente. Este valor varia em função do número de processadores, núcleos e *threads* disponíveis no hardware. Se o servidor possui capacidade, o *Configurator Worker* recupera uma *configurator call* na *Configurator Queue* associada pelo administrador (Figura 4.8, passo 2). Se o *Configurator Worker* verificar que o servidor não possui capacidade, ele aguarda 60s e retorna para o passo 1. Caso não existam *configurator calls* na *Configurator Queue*, o *Configurator Worker* aguarda 5s e retorna para o passo 1.

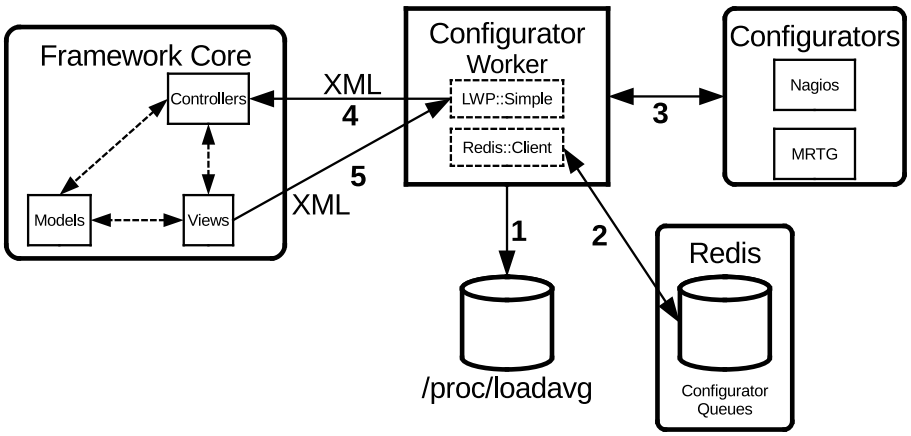


Figura 4.8: Implementação do *Configurator Worker*

A Figura 4.9 apresenta uma *configurator call* recuperada do Redis codificada em JSON (SRIPARASA, 2013). A *configurator call* possui o comando necessário para a execução do *Configurator* com todos os argumentos recuperados do modelo de informação (Figura 4.9, linha 3). O *Configurator Worker* executa este comando e armazena a sua saída e código de *status* (Figura 4.8, passo 3). Estas informações são enviadas através de uma requisição em XML para o REST *Web service* (Figura 4.8, passo 4) utilizando o identificador da *configurator call* (Figura 4.9, linha 2) que foi recuperada da *Configurator Queue*. O REST *Web service* responde à requisição com um XML indicando se a operação foi realizada com sucesso (Figura 4.8, passo 5).

```
1 {"class":"PerformExecutions",
2  "args":[6778,
3          "/usr/lib/FlexACMS/configurator/nagios/nagios_create_host.pl
              --host_name instance-000027c5 --alias instance-000027c5
              --address 192.168.1.130"]
4 }
```

Figura 4.9: Exemplo de uma *configurator call* recuperada do Redis

O código-fonte do *Configurator Worker* possui 92 linhas de código. Seu desenvolvimento é relativamente simples, cuja maior dificuldade está em recuperar as informações sobre as *Configurator Queues* armazenadas no Redis e construir a requisição XML que é enviada para o REST *Web service*.

4.4 *Configurators*

Os *Configurators* são os componentes responsáveis por configurar as soluções de monitoramento para que monitorem as métricas de um *monitoring slice*. Os *Configurators* tratam as peculiaridades das soluções de monitoramento, tais como os seus métodos de configuração. Por exemplo, o Nagios armazena as suas configurações em arquivos, cujo conteúdo pode ser gerado facilmente por um *Configurator*. O MRTG também armazena as suas configurações em arquivos, mas pela complexidade da configuração destes arquivos é mais conveniente gerá-los através dos *scripts* fornecidos pela solução. No decorrer desta seção são detalhados os *Configurators* desenvolvidos para o Nagios e MRTG. Estas soluções são baseadas em software livre, amplamente utilizadas em ambientes tradicionais e que podem ser úteis no contexto de computação em nuvem.

A configuração do Nagios é baseada em objetos e na herança de atributos a partir de *templates* (KOCJAN, 2014). Os objetos *hostgroup* são utilizados para agrupar na interface aqueles dispositivos que compartilham alguma característica, tais como tipo de dispositivo (*e.g.* servidor, máquina virtual, roteador), localização (*e.g.* sede, filial) ou finalidade (*e.g.* produção, desenvolvimento). Os objetos *host* representam o dispositivo monitorado e armazenam informações como seu endereço IP, o *template* que será utilizado e a qual *hostgroup* pertence. Os *templates* simplificam a configuração por evitar que *hosts* semelhantes do ponto de vista do monitoramento tenham informações idênticas repetidas, tais como intervalo de *polling* e método de notificação em caso de falhas. Os objetos *service* representam as métricas monitoradas de um dispositivo e armazenam informações como o *host* a que pertence, o *template* para *service* que será utilizado e o *plugin* do Nagios com os parâmetros necessários para a coleta daquela métrica. Assim como os *templates* para *hosts*, os *templates* para *services* simplificam a configuração das métricas monitoradas por evitar que informações comuns entre *services* tenham que ser repetidas.

Foram desenvolvidos três *Configurators* para tratar de maneira distinta os objetos de configuração *hostgroup*, *host* e *service* do Nagios. Cada um destes *Configurators* foi associado a um nível do FlexACMS por analogia entre suas definições. Por exemplo, o nível *Cloud* representa conjuntos de *cloud slices* para o *framework*, assim como *hostgroups* são conjuntos de *hosts* para o Nagios. Por este motivo, o nível *Cloud* foi associado aos *hostgroups* no Nagios. Ou seja, quando um objeto do nível *Cloud* é criado no *framework*, também será criado um *hostgroup* no Nagios para agrupar os *cloud slices* associados a esta *Cloud*. De maneira semelhante, os *hosts* e *services* do Nagios foram associados aos níveis *Slice* e *Resource*, respectivamente. Estas associações definem os interesses para cada um dos *Configurators* e por consequência as operações detectadas pelo *framework* (*e.g. new cloud*, *new slice*, *new resource*) que exigirão a execução destes *Configurators*.

Na Figura 4.10, as setas identificadas por números indicam interações relacionadas ao processo de configuração do Nagios pelos *Configurators* que foram desenvolvidos utilizando a linguagem Perl. Este processo é semelhante para os três *Configurators* desenvolvidos, modificando apenas o tipo de objeto de configuração que será criado no Nagios. Os *Configurator Workers* são responsáveis por executar estes *Configurators* fornecendo como parâmetros as informações necessárias (Figura 4.10, passo 1). O *Configurator* utiliza os parâmetros recebidos para gerar um arquivo de configuração que será armazenado no diretório de configurações do Nagios (Figura 4.10, passo 2). Para que a nova configuração entre em vigor, o *Configurator* envia um sinal para que o processo principal do Nagios recarregue suas configurações (Figura 4.10, passo 3). O processo principal do Nagios recarrega todas as informações armazenadas em seu diretório de arquivos de configuração (Figura 4.10, passo 4).

Na Figura 4.10, as setas identificadas por letras indicam as interações relacionadas ao funcionamento do Nagios, que serão apresentadas para contextualizar as tarefas realizadas pelo *Configurator*. As configurações armazenadas no diretório de configurações informam ao Nagios quais métricas monitorar e quais os respectivos *plugins* que devem ser utilizados para coletá-las. A partir destas configurações, o Nagios executa os *plugins* para coletar as métricas associadas aos *hosts* e *services* (Figura 4.10, passo a). Para coletar estas métricas, os *plugins* utilizam protocolos específicos para comunicar-se com os dispositivos (*e.g. Simple Network Management Protocol* (SNMP) (HARRINGTON; PRESUHN; WIJNEN, 2002), *Nagios Remote Plugin Executor* (NRPE) (KOCJAN, 2014), *Internet Control Message Protocol* (ICMP) (POSTEL, 1981)) (Figura 4.10, passo b).

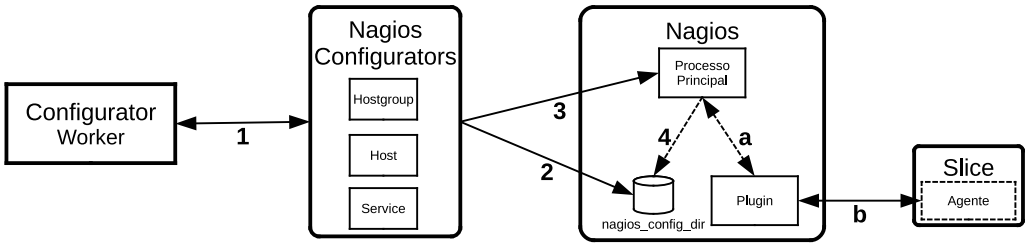


Figura 4.10: Implementação dos *Configurators* desenvolvidos para o Nagios

A configuração do MRTG também é baseada em arquivos de configuração. Entretanto, o conteúdo destes arquivos é complexo por se tratar de um *template* para a página HTML que será gerada. Além disso, possui informações sobre o dispositivo como contatos e sobre suas interfaces como descrição, tipo e taxa de transmissão. Por isso, os arquivos de configuração do MRTG são normalmente gerados com o auxílio de *scripts* fornecidos pela solução. Entre estes *scripts* está o *cfgmaker* que comunica-se com o dispositivo que será monitorado para coletar estas informações, entre outras, e evita que o administrador tenha que construir o arquivo de configuração manualmente. Outro *script* é o *indexmaker* que processa as informações dos arquivos de configuração gerados pelo *cfgmaker* e gera os arquivos HTML que permitirão o acesso às métricas monitoradas.

O MRTG pode ser executado como um *daemon*, pode ser chamado manualmente pelo usuário ou pode ser invocado continuamente através de um processo como a *Cron*. A *Cron* é a opção mais apropriada em ambientes onde a configuração será modificada com frequência, pois evita que o *daemon* precise ser reiniciado para recuperar novas configurações. Entretanto, deve-se adicionar uma entrada na *Cron* com a chamada do MRTG que aponta para o novo arquivo de configuração. Pode-se utilizar um diretório para armazenar todos os *scripts* relacionados ao MRTG e adicionar apenas uma entrada na *Cron* para executar todos *scripts* daquele diretório no intervalo de tempo estipulado pelo administrador. Desta forma, evita-se alterar a configuração da *Cron* a cada nova configuração.

O *Configurator* para o MRTG foi desenvolvido na linguagem Bash (NEWHAM; ROSENBLATT, 2005) e executa os *scripts* fornecidos pelo MRTG para gerar os arquivos de configuração. Este *Configurator* foi associado ao nível *Slice* do FlexACMS para ser executado toda vez que um novo *cloud slice* for criado. Desta forma, o MRTG irá monitorar todas as interfaces de rede deste novo *cloud slice*.

Na Figura 4.11, as setas identificadas por números indicam interações relacionadas ao processo de configuração do MRTG pelo *Configurator*. O *Configurator Worker* executa o *Configurator* passando como parâmetros o nome, a comunidade SNMP e o endereço IP do *cloud slice* (Figura 4.11, passo 1). O *Configurator* executa o *script cfgmaker* (Figura 4.11, passo 2) que entra em contato com o agente SNMP no *cloud slice* para obter informações

sobre as suas interfaces de rede (Figura 4.11, passo 3). De posse destas informações, o *cfgmaker* cria um arquivo de configuração para o *cloud slice* no diretório de arquivos de configurações do MRTG (Figura 4.11, passo 4). A partir deste momento, o *Configurator* executa o *script indexmaker* (Figura 4.11, passo 5) que utiliza as informações contidas no arquivo de configuração criado pelo *cfgmaker* (Figura 4.11, passo 6) para gerar os arquivos HTML no diretório associado ao *vhost* do MRTG no Apache (Figura 4.11, passo 7). O *Configurator* cria um *script* na linguagem Bash que executa o MRTG apontando para o arquivo de configuração gerado pelo *cfgmaker* e o armazena no diretório de *scripts* do MRTG na *Cron* (Figura 4.11, passo 8).

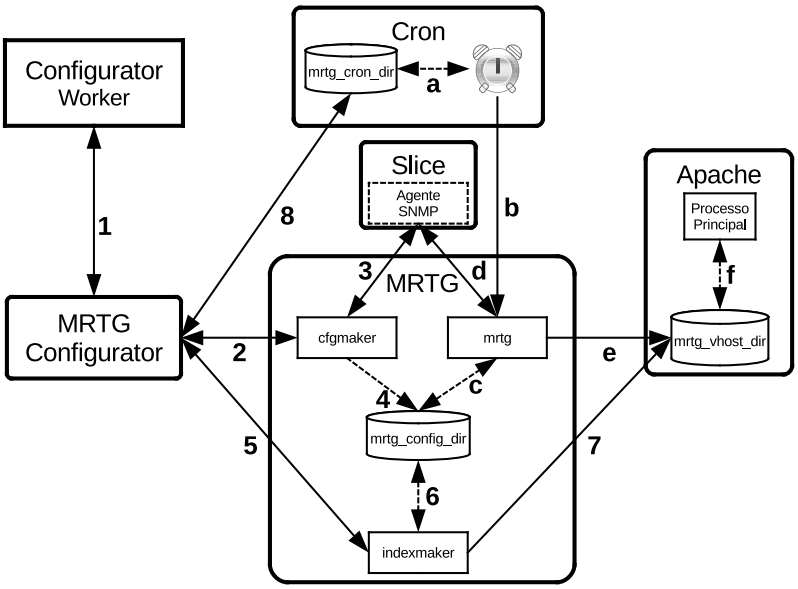


Figura 4.11: Implementação do *Configurator* desenvolvido para o MRTG

Na Figura 4.11, as setas identificadas com letras indicam interações relacionadas ao funcionamento do MRTG para contextualizar as tarefas realizadas pelo *Configurator*. A *Cron* executa os *scripts* armazenados no diretório de *scripts* do MRTG em intervalos de tempo definidos pelo administrador (Figura 4.11, passo a). Este diretório possui, para cada *cloud slice* monitorado, um *script* que executa o programa *mrtg* (Figura 4.11, passo b) apontando para o arquivo de configuração correspondente ao *cloud slice* que será monitorado (Figura 4.11, passo c). O programa *mrtg* coleta as métricas sobre a utilização das interfaces de rede através do agente SNMP instalado no *cloud slice* (Figura 4.11, passo d) e atualiza os arquivos que fornecem estas métricas para os usuários (Figura 4.11, passo e). Estes arquivos são acessados pelos usuários através do Apache (Figura 4.11, passo f).

A Tabela 4.2 mostra o número de linhas de código utilizadas para desenvolver cada *Configurator*. O esforço de desenvolvimento para cada *Configurator* é semelhante conforme indica o número de linhas de código. Pode-se perceber que mesmo soluções de monitoramento que são mais complexas para serem configuradas, como o MRTG, podem não propagar esta complexidade para o desenvolvimento de seu *Configurator*. No caso do MRTG, a complexidade foi reduzida pela utilização dos *scripts* fornecidos pela solução de monitoramento. Para o Nagios, o esforço de desenvolvimento está em gerar arquivos de configuração conforme a sintaxe exigida pelo Nagios e receber as informações que são necessárias como parâmetros. Esta facilidade em desenvolver *Configurators* é um ponto forte para o *framework*, pois facilita que os próprios administradores desenvolvam seus *Configurators* para atender as suas necessidades.

Tabela 4.2: Número de linhas de código necessárias para codificar cada *Configurator*

Configurator	**Número de linhas de código**
`nagios_create_hostgroup.pl`	32 linhas
`nagios_create_host.pl`	53 linhas
`nagios_create_service.pl`	47 linhas
`mrtg_configurator.sh`	44 linhas

Para a utilização do FlexACMS pode ser necessário que os administradores de ambientes de computação em nuvem tenham que desenvolver *Gatherers* e *Configurators*. O FlexACMS permite que qualquer linguagem seja utilizada para desenvolver estes componentes, o que também facilita a adoção e customização do *framework*. *Gatherers* podem ser desenvolvidos para compatibilizar novas plataformas com o *framework* ou para integrar sistemas utilizados pelos provedores. Por exemplo, pode-se desenvolver um *Gatherer* que consulte as informações sobre as assinaturas de *Monitoring as a Service* dos usuários e complementar as informações que foram recuperadas da plataforma. Por sua vez, novos *Configurators* podem ser necessários para que o *framework* seja capaz de configurar outras soluções de monitoramento. Cabe salientar que estes componentes podem ser implementados e disponibilizados por terceiros para serem reutilizados em outros ambientes. Desta forma, o desenvolvimento de componentes do *framework* oferece vantagem em relação ao desenvolvimento de *scripts* de automação, pois estes *scripts* geralmente são específicos para os ambientes no qual foram desenvolvidos.

Para customização e aprimoramento do *Framework Core*, o desenvolvedor deverá possuir um bom entendimento do *framework Ruby on Rails*. Deve-se entender os relacionamentos entre os componentes do MVC e das convenções adotadas pelo *framework*. Neste sentido, a leitura da Seção 4.1 é fundamental como ponto de partida para o entendimento do *Ruby on Rails*, porém será necessário buscar informações mais detalhadas.

No decorrer deste capítulo foram apresentados detalhes de implementação dos componentes do protótipo FlexACMS. Apenas o *Framework Core* da arquitetura estendida foi apresentado, pois esta é a arquitetura sugerida como solução para a criação automatizada de *monitoring slices*. Entretanto, no próximo capítulo são apresentadas avaliações realizadas com as arquiteturas inicial e estendida com o intuito de mostrar e medir ganhos e benefícios introduzidos pela arquitetura estendida.

5 AVALIAÇÃO

Neste capítulo são apresentados os experimentos conduzidos com o intuito de avaliar as arquiteturas propostas para o FlexACMS. Tais experimentos avaliam o tempos de resposta dos componentes das arquiteturas inicial e estendida, o consumo de banda de comunicação exigido pelos *Gatherers* e o consumo de processamento e memória da arquitetura inicial. O objetivo de cada um destes experimentos é detalhado no decorrer deste capítulo, mas são norteados para a avaliação da escalabilidade e viabilidade do FlexACMS. Os experimentos relacionados ao tempo de resposta foram projetados de acordo com a metodologia descrita por Raj Jain (1991) para determinar os fatores que podem influenciar os resultados dos experimentos, para escolher os níveis em que estes fatores serão observados e para selecionar os melhores fatores (projeto 2^k) com base em suas contribuições (efeitos) para a variabilidade dos resultados. Os demais experimentos utilizaram níveis de acordo com o objetivo e cenário de avaliação utilizado.

Na Seção 5.1 é apresentado o experimento que avalia o tempo de resposta da arquitetura inicial do FlexACMS. Este experimento demonstra que o desempenho desta arquitetura é inadequado e não atinge escalabilidade em relação ao número de *cloud slices* armazenados no *framework*. Na Seção 5.2 é apresentado o experimento que avalia o consumo de processamento e memória da arquitetura inicial. Este experimento demonstra que o consumo de recursos da arquitetura inicial é adequado, porém também demonstra que esta arquitetura não utiliza plenamente os recursos disponíveis. Estas observações foram fundamentais para identificar os problemas da arquitetura inicial e elaborar as melhorias que foram introduzidas na arquitetura estendida.

Na Seção 5.3 é realizado um comparativo dos tempos de resposta das arquiteturas para avaliar se as melhorias introduzidas no *framework* reduzem o tempo de resposta. Além disso, este experimento foi conduzido utilizando *Gatherers* e *Configurators* reais, ou seja, este experimento mostra o desempenho do *framework* em um ambiente real de provedores de computação em nuvem. Na Seção 5.4 é avaliado o consumo de banda de comunicação exigido pelo *Gatherer* desenvolvido para a OpenStack API. Na Seção 5.5 são apresentados os tempos de resposta dos *Configurators* desenvolvidos para o Nagios e MRTG. Na Seção 5.6 é apresentado o experimento que observa o tempo de resposta da arquitetura estendida em relação ao número de métricas por *monitoring slice*, número de *monitoring slices* construídos em uma rajada e número de *cloud slices* armazenados no *framework*. Além disso, os resultados deste experimento são utilizados para analisar a escalabilidade do FlexACMS em relação ao número de *cloud slices* armazenados no *framework*. O consumo de processamento e memória da arquitetura estendida não foi avaliado, pois estes parâmetros podem ser ajustados de acordo com as necessidades do administrador através da quantidade de trabalhadores que são empregados.

5.1 Tempo de resposta da arquitetura inicial

O objetivo deste experimento é avaliar o tempo de resposta necessário para que o *Framework Core* configure as soluções de monitoramento após receber um conjunto de informações dos *Gatherers*. Os tempos de resposta obtidos foram analisados para avaliar a escalabilidade do FlexACMS em relação ao número de *cloud slices* armazenados no *framework*. Além disso, outro fator importante que foi analisado é o número de *monitoring slices* que deverão ser criados a partir de uma única requisição do *Gatherer*, ou seja, o número de *monitoring slices* criados em uma rajada.

Este experimento foi realizado quando apenas o *Framework Core* baseado na arquitetura inicial havia sido desenvolvido. Portanto, outros componentes como *Gatherers* e *Configurators* foram substituídos por programas que simulam seu funcionamento. Neste sentido, foram desenvolvidos dois tipos de *Gatherers*: simples e eficiente, com o intuito de avaliar o impacto dos *Gatherers* no tempo de resposta do *framework*. As soluções de monitoramento e seus respectivos *Configurators* foram substituídos por um *Configurator* que utiliza uma conexão SSH para criar um arquivo no servidor de monitoramento. Desta forma, basta verificar se o arquivo foi criado para garantir que o FlexACMS processou a requisição. Este processo simula a criação de um arquivo de configuração para cada *cloud slice*, ou seja, o *monitoring slice* correspondente é composto por uma métrica.

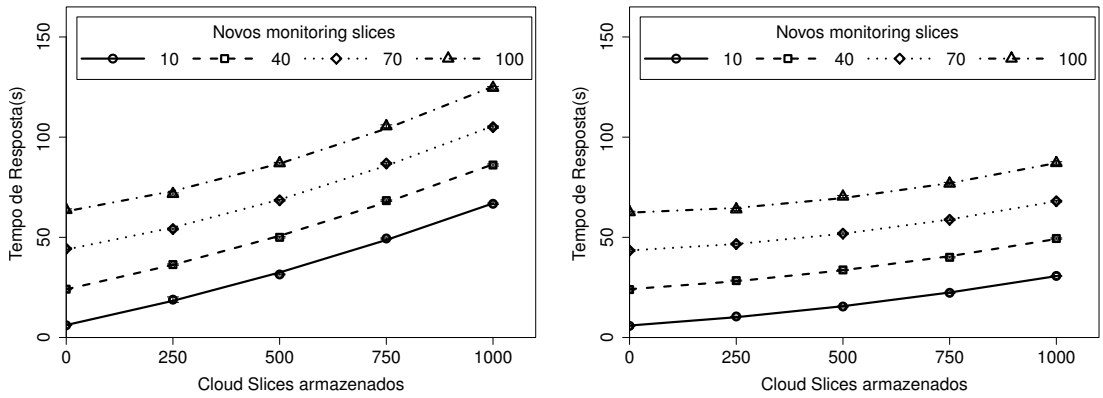
O *Gatherer* simples recupera todas as informações sobre a plataforma de nuvem e envia para o *Framework Core*. Este *Gatherer* é simples porque apenas converte a informação recebida da plataforma para o formato aceito pelo REST *Web service*. O *Gatherer* eficiente controla quais informações já foram enviadas para o *Framework Core* e envia apenas as modificações ou novas informações adicionadas à plataforma. Neste experimento, o *Gatherer* eficiente pode ser comparado aos *Gatherers* desenvolvidos para plataformas que possuem alguma facilidade para descobrir a criação de novos *cloud slices*. Neste caso, o *Gatherer* poderia receber uma mensagem da plataforma informando a criação de novos *cloud slices* ou poderia processar o *log* de execução da plataforma em busca destas operações. A partir destas informações, o *Gatherer* eficiente enviaria para o *Framework Core* apenas informações sobre os novos *cloud slices*.

Foram realizadas observações utilizando cenários de ambientes de computação em nuvem com 0, 250, 500, 750 e 1000 *cloud slices* armazenados no *framework*. Em cada um destes cenários, o *Framework Core* foi avaliado com a adição de 10, 40, 70 e 100 novos *cloud slices*, que exigem a configuração dos *monitoring slices* correspondentes. Apesar dos níveis destes fatores serem suficientes para as conclusões obtidas nesta seção, eles foram ajustados para a avaliação de escalabilidade da arquitetura estendida apresentada na Seção 5.6. Foram utilizados os *Gatherers* simples e eficiente descritos anteriormente, sendo que informações sobre os novos *cloud slices* são posicionadas aleatoriamente, em meio às informações dos *cloud slices* antigos, nas requisições geradas pelo *Gatherer* simples para que o posicionamento não influencie no tempo de resposta.

O cenário de avaliação foi composto por 2 servidores com um processador Intel(R) Core(TM) i5 CPU 650@3.20 GHz, 4GB de memória RAM, executando o sistema operacional Gentoo Linux no servidor que hospeda o monitoramento e os *Gatherers*, e executando o sistema operacional Ubuntu Server 12.04 LTS no servidor que hospeda os demais componentes do *framework*. Ambos servidores foram conectados a um switch com enlaces de 1 Gbps de capacidade. Foi utilizado um servidor para hospedar tanto o *Gatherer* como o monitoramento para simplificar a medição do tempo de resposta utilizando o relógio interno da máquina, pois são evitados problemas relacionados à sincronização dos relógios. Em cenários reais, estes componentes provavelmente estariam instalados em

servidores diferentes. Cada observação foi repetida 30 vezes e os gráficos mostram intervalos com 95% de nível de confiança.

As Figuras 5.1a e 5.1b mostram o tempo de resposta para cada cenário quando são utilizados os *Gatherers* simples e eficiente, respectivamente, para enviar informações da plataforma de nuvem para o *Framework Core*. Como o *Gatherer* simples envia mais informações para o *Framework Core*, é esperado que tenha um tempo de resposta mais alto. De fato, o tempo de resposta adicional é causado pelo processamento adicional exigido para tratar as informações enviadas pelo *Gatherer* simples no *Framework Core*. A partir desta observação, pode-se concluir que os *Gatherers* influenciam no tempo de resposta do *framework*. Entretanto, como discutido anteriormente, o desenvolvimento de *Gatherers* eficientes pode depender de características da plataforma de nuvem como *log* de execução ou algum tipo de mecanismo de troca de mensagens que informe ao *Gatherer* as informações que foram inseridas ou modificadas na plataforma.



(a) Arquitetura inicial - *Gatherer* simples

(b) Arquitetura inicial - *Gatherer* eficiente

Figura 5.1: Tempo de resposta da arquitetura inicial com *Gatherers* simples e eficiente

Para avaliar a escalabilidade da arquitetura inicial, estes resultados foram analisados através de regressões para modelos lineares e não-lineares. Esta análise visa verificar a complexidade da solução em relação ao número de *cloud slices* armazenados (n). Foram verificadas as seguintes complexidades: $O(n)$, $O(n^2)$, $O(n^3)$ e $O(e^n)$ para cada conjunto de pontos relacionados a criação de um mesmo número de novos *monitoring slices* em cada uma das Figuras 5.1a e 5.1b.

A Tabela 5.1 mostra as médias para os parâmetros estatísticos dos modelos obtidos pelas regressões para os conjuntos de pontos de uma mesma figura. Estes parâmetros e as regressões foram obtidos utilizando a função `lm` do software R (R Core Team, 2014) que utiliza a distribuição *Student's t* (JAIN, 1991) para calcular o *p-value* ($Pr(> |t|)$). O coeficiente de determinação (R^2) indica quanto da variabilidade dos resultados é explicada pelo modelo obtido pela regressão (JAIN, 1991). Já o parâmetro *p-value* indica a probabilidade de que um termo obtido pelo modelo seja nulo, o que indica a inadequação do modelo para representar os resultados obtidos. Ao analisar estes parâmetros na Tabela 5.1, pode-se afirmar que o modelo para a complexidade $O(n^2)$ é o mais adequado para representar os resultados obtidos. O modelo para a complexidade $O(n^2)$ possui parâmetro R^2 semelhante ao alcançado por $O(n^3)$ e probabilidade baixa de que o termo β_2 seja nulo. Embora o modelo para $O(n^3)$ apresente o melhor resultado para R^2, este modelo apresenta o parâmetro *p-value* para o termo de mais alta ordem (β_3) bastante elevado,

indicando que o modelo pode ser inadequado para representar os resultados obtidos. As curvas obtidas pelo modelo para a complexidade $O(n^2)$ estão ilustradas nas Figuras 5.1a e 5.1b sobre os pontos reais obtidos no experimento. Pode-se observar nas figuras que as curvas obtidas pelos modelos representam com precisão os resultados obtidos, conforme indica a análise dos parâmetros estatísticos R^2 e *p-value*.

Tabela 5.1: Média aritmética dos parâmetros estatísticos das regressões lineares e não-lineares dos tempos de resposta da arquitetura inicial com *Gatherers* simples e eficiente

Complex.	**Modelo**	**Simples**		**Eficiente**	
		R^2	*p-value*	R^2	*p-value*
$O(n)$	$y = \beta_0 + \beta_1 x$	0,98770	0,00062	0,96685	0,00283
$O(n^2)$	$y = \beta_0 + \beta_1 x + \beta_2 x^2$	0,99895	0,04526	0,99875	0,01893
$O(n^3)$	$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$	0,99968	0,46612	0,99928	0,44008
$O(e^n)$	$y = \beta_0 + \beta_1 e^x$	0,97882	0,00207	0,98100	0,00138

Conclui-se então que a complexidade da arquitetura inicial do *framework* pode ser aproximada por $O(n^2)$. Entretanto, esta complexidade afeta negativamente a escalabilidade da arquitetura inicial do *framework*, pois indica que o tempo de resposta do *framework* varia quadraticamente em relação ao tamanho do ambiente (número de *cloud slices* armazenados). Este resultado demonstra que a arquitetura inicial é inadequada e que precisa ser melhorada para alcançar escalabilidade. Estas melhorias foram introduzidas na arquitetura estendida, conforme discutido na Seção 3.3.

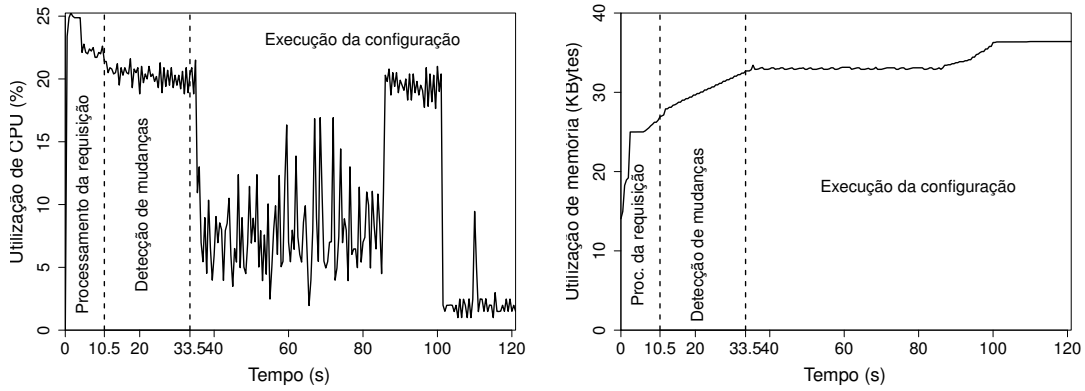
5.2 Consumo de processamento e memória da arquitetura inicial

O objetivo deste experimento é avaliar se o FlexACMS exige servidores de grande capacidade para ser utilizado. A escalabilidade da solução pode ser prejudicada se ela exigir grande expansão de capacidade a medida que o ambiente monitorado aumenta. A utilização de processamento e memória foi observada em cada fase do tratamento de uma requisição, *i.e.*, processando a requisição, detectando mudanças e executando a configuração. A requisição com o maior tempo de resposta no experimento anterior foi escolhida para ser observada. Esta requisição foi gerada por um *Gatherer* simples enviando informações sobre 1000 *cloud slices* antigos com a adição de 100 *cloud slices* novos.

Os valores sobre utilização de processamento e memória foram coletados diretamente do diretório `/proc`, mesma fonte de informações utilizada pelas ferramentas `ps` e `top`. O `ps` gera um percentual de utilização de processamento sobre o tempo total de vida do processo, mas a intenção deste experimento é coletar valores instantâneos de utilização de processamento e memória. O `top` gera um percentual de utilização de processamento desde a última atualização da interface do usuário. Portanto, precisa-se informar ao `top` a frequência que os resultados devem ser enviados para a interface do usuário e quantas coletas devem ser executadas em uma execução do `top`. Como não se sabe quantas coletas o `top` deve executar em cada fase do processamento da requisição, um *script* foi desenvolvido que é capaz de saber que fase do processamento da requisição está sendo executada. Este *script* coleta valores de processamento e memória a uma frequência de 0,5 segundo dos arquivos `/proc/PID/stat` e `/proc/stat`. Esta frequência é suficiente para as observações realizadas nesta seção.

Neste experimento foi utilizado o mesmo cenário de avaliação descrito na Seção 5.1. A Figura 5.2a mostra os resultados para utilização de processamento. Na arquitetura inicial, todo o processamento da requisição é realizado por apenas uma *thread*. Ou seja, a arquitetura inicial pode utilizar até 25% da capacidade de processamento do servidor utilizado neste experimento, cujo processador é capaz de executar 4 *threads* simultaneamente porque possui 2 *cores* capazes de executar simultaneamente 2 *threads* cada.

Na fase de processamento da requisição (0s-10,5s), a arquitetura inicial utiliza praticamente a capacidade total da *thread* (25%). Nesta fase, o XML enviado pelo *Gatherer* é verificado e armazenado na base de dados do *framework*. Estas tarefas foram limitadas por CPU (*CPU-bound*), como ilustrado na Figura 5.2a, que mostra consumo alto de processamento durante estas tarefas. Mesmo a tarefa de armazenamento, que impactaria mais sobre a E/S do banco de dados do *framework*, foi bastante limitada pela CPU.



(a) Utilização de processamento

(b) Utilização de memória

Figura 5.2: Utilização de processamento e memória da arquitetura inicial

Na fase de detecção de mudanças (10,5s-33,5s), as informações recebidas pelo REST *Web service* sobre os *cloud slices* são verificadas para detectar se elas referem-se a *cloud slices* novos ou antigos. Se a informação refere-se a *cloud slices* novos, esta informação é armazenada no banco de dados do *framework* na base "*Processed Information*"; caso contrário, a informação é descartada. Quando uma nova informação é detectada, uma mudança é inserida em uma tabela de mudanças. Todas estas tarefas recaem sobre o banco de dados do *framework* e possui utilização de processamento semelhante àquela observada na tarefa de armazenamento da fase de processamento da requisição.

Na fase de execução da configuração (33,5s-122s) pode-se observar três comportamentos diferentes que estão relacionados à maneira que a tabela de mudanças é organizada e como o módulo *Configuration Executor* consulta esta tabela. Neste experimento são enviadas informações sobre 1000 *cloud slices* antigos juntamente com 100 *cloud slices* novos, simulando a adição de 100 *cloud slices* novos na plataforma. Portanto, os três comportamentos observados correspondem, respectivamente, às seguintes situações: mudanças já processadas anteriormente que são de interesse dos *Configurators*, mas que foram configuradas no passado; mudanças sobre novos *cloud slices* que exigem a criação de *monitoring slices* (alta utilização de processamento para configurar as soluções de monitoramento); e espera para receber as respostas dos *Configurators*.

A Figura 5.2b mostra a utilização de memória em cada fase do processamento da requisição. A utilização de memória aumenta rapidamente na fase de processamento da requisição porque diversas estruturas de dados são criadas para verificar o XML recebido

pelo REST *Web service*. Depois de verificar o documento XML, as informações recebidas são armazenadas em objetos que são instâncias das classes internas do protótipo. Na fase de detecção de mudanças, o consumo de memória aumenta lentamente de acordo com as mudanças que são detectadas. Para cada mudança detectada, é criado um objeto que representa esta mudança, portanto consumindo uma certa quantidade de memória. Durante a fase de execução da configuração, a utilização permanece constante durante quase todo o período. O consumo de memória aumenta durante o segundo comportamento observado na utilização de processamento nesta mesma fase, que corresponde à configuração das soluções de monitoramento. Durante a configuração das soluções de monitoramento, a arquitetura inicial instancia objetos para armazenar a saída e *status* de execução dos *Configurators* aumentando o consumo de memória nesta fase.

Ao analisar os resultados de utilização de processamento e memória pode-se concluir que a arquitetura inicial tem uma utilização de recursos adequada. A utilização de processamento é alta (em torno de 20%) durante 50,5s (33,5s+17s) para um processamento total que leva 122s, o que é aceitável para processar uma requisição com 1100 *cloud slices*. Em termos de utilização de memória, a arquitetura inicial utiliza 40 Kbytes no pico, o que é um percentual aceitável (0,01%) da memória disponível no servidor. Entretanto, pode-se observar que a arquitetura inicial não utiliza plenamente os recursos disponíveis, principalmente em relação à utilização de processamento. Esta arquitetura utiliza apenas uma *thread* no tratamento de uma requisição, apesar de o processador do servidor ser capaz de executar 4 *threads* simultaneamente. Isto acontece porque a arquitetura inicial executa sequencialmente todas as fases do processamento da requisição, ou seja, o processamento da requisição é modelado como uma tarefa única.

5.3 Comparativo dos tempos de resposta das arquiteturas inicial e estendida

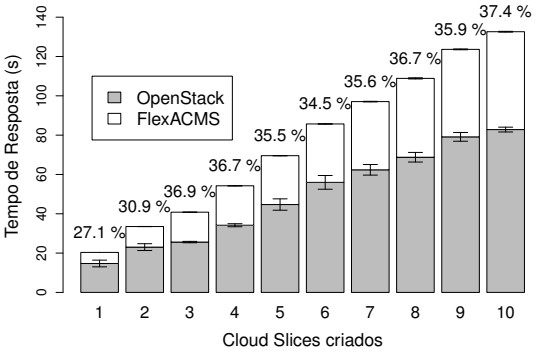
Este experimento foi realizado para alcançar dois objetivos. O primeiro objetivo consiste em avaliar o comportamento do FlexACMS ao se utilizar todos os componentes reais: *Framework Core*, *Gatherers*, *Configurators* e trabalhadores no caso da arquitetura estendida. Nesta avaliação, é possível verificar se o tempo de resposta do FlexACMS é adequado ao tempo de resposta da plataforma, ou seja, se estão na mesma ordem de magnitude. O segundo objetivo consiste em comparar os ganhos que a arquitetura estendida alcança sobre a arquitetura inicial, ou seja, se as modificações introduzidas na arquitetura estendida alcançam de fato melhores resultados.

Os tempos de resposta do OpenStack e do FlexACMS foram observados para avaliar se o tempo de resposta do *framework* é adequado ao tempo de resposta da plataforma. No OpenStack foi avaliado o tempo exigido para criar um conjunto de *cloud slices* na plataforma. No FlexACMS foi contabilizado o tempo correspondente para enviar informações sobre os novos *cloud slices* e configurar os *monitoring slices* exigidos pelos novos *cloud slices*. Neste experimento, os *monitoring slices* são compostos por duas métricas que serão monitoradas pelo Nagios: estado do *host* e utilização de CPU. Portanto, foram cadastrados no *framework* dois *Configurators* do Nagios: um para a configuração do monitoramento do *host* e outro para a utilização de CPU.

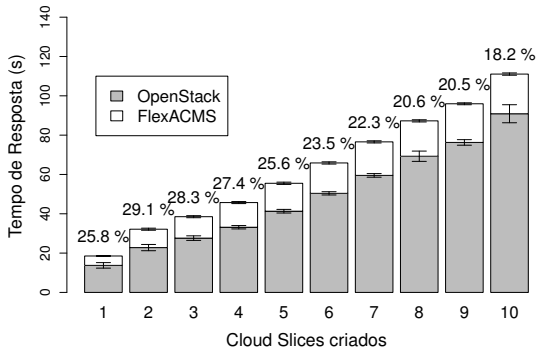
O cenário de avaliação é semelhante ao ambiente de um provedor de nuvens IaaS. Este cenário foi composto por dois servidores: um servidor para a plataforma OpenStack e outro servidor para o monitoramento. O servidor da plataforma possui dois processadores Intel(R) Xeon(R) CPU E5430@2.66GHz com 4 *cores* cada, 16GB de memória RAM,

utilizando o sistema operacional Ubuntu Server 12.04 LTS. O servidor de monitoramento possui um processador Intel(R) Core(TM) i5 CPU 650@3.20GHz com 2 *cores*, 4GB de memória RAM, utilizando o sistema operacional Ubuntu Server 12.10. Ambos estão conectados a um switch por enlaces de 1 Gbps de capacidade. O servidor da plataforma hospeda o *Gatherer* para a OpenStack API, descrito na Seção 4.2, e todos os serviços e componentes da infraestrutura do OpenStack. O servidor de monitoramento hospeda o *Framework Core*, as soluções de monitoramento Nagios e MRTG, seus respectivos *Configurators* e, quando observa-se a arquitetura estendida, também executa 10 *Configurator Workers* simultaneamente. Cada observação foi repetida 30 vezes e os gráficos mostram intervalos com 95% de nível de confiança.

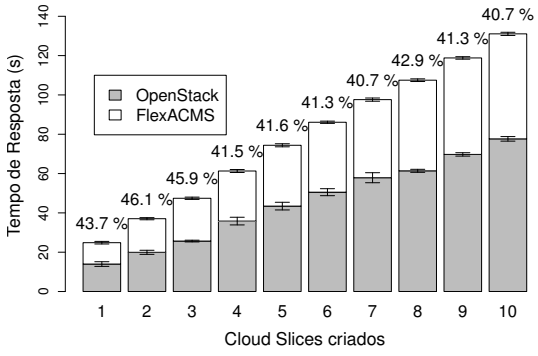
A Figura 5.3a mostra os resultados da avaliação do tempo de resposta para a arquitetura inicial. O OpenStack exige de 73-63% do tempo total do experimento para criar *cloud slices*. A arquitetura inicial exige de 37-27% do tempo total do experimento para configurar os *monitoring slices* correspondentes. A Figura 5.3b mostra os resultados do tempo de resposta da arquitetura estendida. Neste experimento, o OpenStack exige de 82-71% do tempo total do experimento para criar *cloud slices*, enquanto a arquitetura estendida exige de 29-18% do tempo total do experimento para configurar os *monitoring slices* correspondentes. Portanto, a arquitetura estendida reduz em torno de 10% o impacto da configuração automática de *monitoring slices* no tempo total do experimento.



(a) Arquitetura inicial (2 métricas)



(b) Arquitetura estendida (2 métricas)



(c) Arquitetura estendida (52 métricas)

Figura 5.3: Comparativo do tempo de resposta das arquiteturas inicial e estendida

A alta variação do tempo de resposta do OpenStack, como indicado pelos intervalos de confiança, prejudica a análise do tempo total do experimento. Para contornar esta variação, pode-se comparar os tempos de resposta em valores absolutos obtidos pelas arquiteturas inicial e estendida. A Tabela 5.2 apresenta estes valores e o ganho percentual da arquitetura estendida sobre a arquitetura inicial. Pode-se observar que o tempo de resposta da arquitetura inicial aumenta em torno de 5s para cada novo *monitoring slice*. Para a arquitetura estendida este aumento fica em torno de 1s e 2s. Ou seja, o tempo de resposta da arquitetura estendida não aumenta significativamente ao se elevar o número de *monitoring slices* configurados. Esta constatação pode ser confirmada ao se observar os ganhos percentuais do tempo de resposta da arquitetura estendida sobre a arquitetura inicial. Este ganho percentual chega a 59,4% para a configuração de 10 *monitoring slices*. Neste caso, o ganho é maior porque existem mais tarefas de configuração para serem executadas e, portanto, existem mais possibilidades para execução de tarefas em paralelo reduzindo o tempo de resposta. Estas observações mostram que a arquitetura estendida é adequada quando um grande número de *monitoring slices* precisa ser criado.

Tabela 5.2: Tempos de resposta em valores absolutos e ganhos da arquitetura estendida sobre a arquitetura inicial

Monitoring Slices **configurados**	**Arquitetura**		**Ganho**
	Inicial	**Estendida**	
1	5,63s	4,78s	15,1%
2	10,47s	9,35s	10,7%
3	15,19s	10,90s	28,2%
4	20,06s	12,55s	37,4%
5	24,85s	14,20s	42,8%
6	29,71s	15,46s	48,0%
7	34,69s	17,05s	50,8%
8	40,11s	18,00s	55,1%
9	44,50s	19,66s	55,8%
10	49,77s	20,17s	59,4%

Os *monitoring slices* configurados nas observações ilustradas nas Figuras 5.3a e 5.3b são compostos por duas métricas: estado do *host* e utilização de CPU. Entretanto, é necessário observar a arquitetura estendida na configuração de *monitoring slices* mais complexos: estado do *host*, 50 métricas monitoradas pelo Nagios e um gráfico de utilização de rede construído pelo MRTG. Neste caso, cada *monitoring slice* é composto por 52 métricas e exige 52 *configurator calls* para serem configurados.

A Figura 5.3c mostra que a arquitetura estendida exige um tempo maior para configurar *monitoring slices* mais complexos. Neste cenário, o OpenStack exige em torno de 60-55% do tempo do experimento para criar os *cloud slices* contra 45-40% do tempo requerido pela arquitetura estendida para configurar os *monitoring slices* correspondentes. De fato, analisando as barras relacionadas com a criação de 10 *cloud slices* nas Figuras 5.3b e 5.3c pode-se perceber que o tempo de resposta para configurar *monitoring slices* mais complexos é 2,46 vezes (50,4s na Figura 5.3c) maior que o tempo de resposta para configurar *monitoring slices* mais simples (20,2s na Figura 5.3b). Portanto, pode-se concluir que o número de métricas em um *monitoring slice* afeta o tempo de resposta da

arquitetura estendida do FlexACMS. Entretanto, o tempo de resposta e o tamanho dos *monitoring slices* não aumentam na mesma proporção (2,46 *vs.* 26 vezes, respectivamente).

Conclui-se então que a arquitetura estendida melhora os tempos de resposta obtidos pela arquitetura inicial reduzindo a influência da configuração automática dos *monitoring slices* no tempo total dos experimentos. Além disso, verificou-se que os tempos de resposta do FlexACMS são adequados se comparados aos tempos de resposta do OpenStack, ou seja, estão em uma mesma ordem de magnitude. Ao se comparar o tempo de resposta absoluto alcançado pela arquitetura estendida, pode-se perceber que a arquitetura estendida possui um ganho de até 59,4% sobre a arquitetura inicial. No experimento onde são configurados *monitoring slices* com 52 métricas, o tempo de resposta do FlexACMS aumenta a uma proporção menor que o aumento dos *monitoring slices*. Entretanto, continua com tempo de resposta menor do que o tempo de resposta do OpenStack. Portanto, o tempo de resposta da configuração automática de *monitoring slices* é adequado se comparado ao tempo necessário pela plataforma OpenStack para criar os *cloud slices*.

5.4 Consumo de banda de comunicação do *Gatherer* desenvolvido para a OpenStack API

O objetivo deste experimento é avaliar o impacto introduzido pelos *Gatherers* em relação ao consumo de banda de comunicação. É esperado que os *Gatherers* sejam executados através de um processo de *polling* de poucos minutos para que novos *cloud slices* tenham seus *monitoring slices* criados rapidamente. Logo, o impacto na rede de comunicação causado pelos *Gatherers* será repetido continuamente ao longo do tempo. Portanto, um consumo elevado de banda de comunicação pelos *Gatherers* poderia prejudicar o funcionamento da rede e a viabilidade do FlexACMS.

Foi observado o tráfego de gerência gerado pelo *Gatherer* da OpenStack API para enviar informações do OpenStack para o FlexACMS. Foi utilizado o cenário de avaliação apresentado na Seção 5.3 utilizando o software `tcpdump` (OREBAUGH; RAMIREZ; BEALE, 2006) para coletar o tráfego no servidor da plataforma e o software Wireshark (OREBAUGH; RAMIREZ; BEALE, 2006) para analisar os resultados. Cada observação foi repetida 30 vezes e os gráficos mostram intervalos com 95% de nível de confiança.

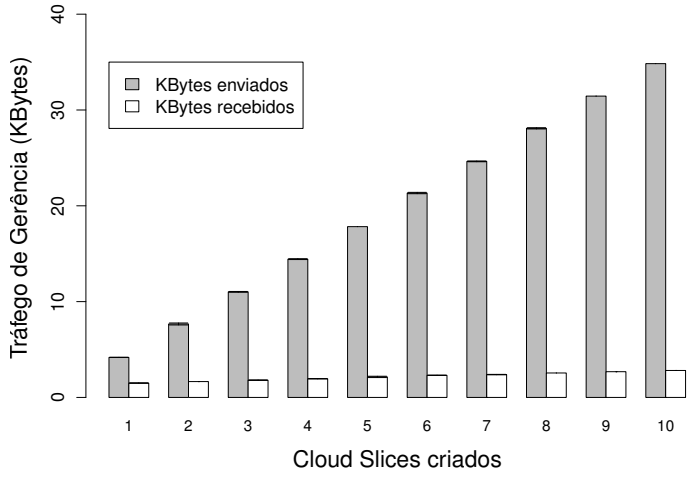


Figura 5.4: Tráfego de gerência para o *Gatherer* da OpenStack API

A Figura 5.4 mostra que o tráfego de gerência recebido pelo servidor da plataforma OpenStack proveniente do servidor de monitoramento é praticamente constante quando um número crescente de *cloud slices* são informados para o *Framework Core*. Isto acontece porque a resposta do REST *Web service* possui apenas a identificação da requisição que foi recebida, cujo o tamanho não varia em função do número de *cloud slices* informados na requisição. Entretanto, quanto mais informações são enviadas para o *Framework Core*, mais pacotes são necessários e devem ser confirmados (TCP ACK - *Transmission Control Protocol Acknowledgement*). O aumento percebido no tráfego recebido está relacionado à confirmação de pacotes enviados.

A Figura 5.4 também mostra o tráfego de gerência enviado pelo servidor da plataforma OpenStack para o servidor de monitoramento. O tráfego aumenta em função do número de *cloud slices* criados como pode ser observado na Figura 5.4. Cada *cloud slice* exige em torno de 4 *Kbytes* de tráfego de gerência. Esta quantidade de tráfego está relacionada à requisição, codificada em XML, necessária para enviar informações para o *Framework Core*. Entretanto, o consumo de banda de comunicação e, consequentemente, o tempo exigido para transmitir esta requisição em uma rede local com capacidade nominal de 1 *Gbps* pode ser considerado insignificante. Logo, pode-se afirmar que o *Gatherer* para a OpenStack API possui consumo de banda adequado.

5.5 Tempo de resposta dos *Configurators* para Nagios e MRTG

O objetivo deste experimento é avaliar o tempo de resposta exigido pelos *Configurators* para configurar as respectivas soluções de monitoramento. Foram observados os tempos de resposta para três *Configurators* descritos na Seção 4.4 utilizados para configuração: do Nagios para *host* e *service* e do MRTG para a criação de gráficos de consumo de rede. O cenário de avaliação utilizado foi o mesmo da Seção 5.3. Cada observação foi repetida 30 vezes e o gráfico mostra intervalos com 95% de nível de confiança.

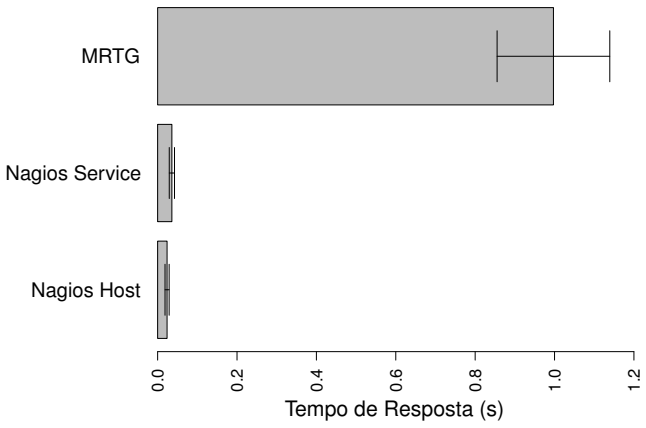


Figura 5.5: Tempo de resposta dos *Configurators* para Nagios e MRTG

A Figura 5.5 mostra os resultados obtidos neste experimento para os três *Configurators* observados. Pode-se perceber que o tempo de resposta do *Configurator* para o MRTG é maior do que o tempo de resposta para os *Configurators* do Nagios. Conforme descrito na Seção 4.4, o *Configurator* para o MRTG utiliza o *script cfgmaker* que comunica-se com o objeto a ser monitorado para obter informações sobre as suas interfaces de rede.

Esta comunicação aumenta o tempo de resposta do *Configurator* e também influencia os intervalos de confiança por introduzir variação nos tempos de resposta. Por outro lado, os *Configurators* para o Nagios utilizam as informações recebidas do *Framework Core* como parâmetros para criar os arquivos de configuração. O *Configurator* para *service* no Nagios utiliza mais informações e gera arquivos de configuração maiores, o que explica a diferença no tempo de resposta se comparado ao *Configurator* para *host* no Nagios.

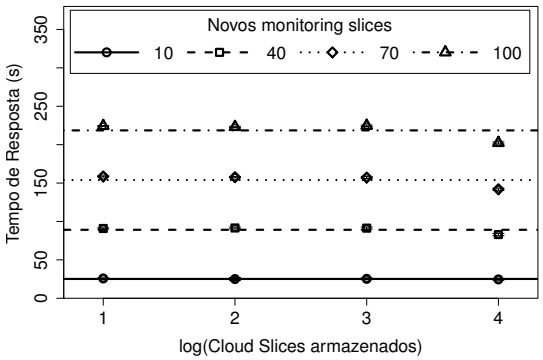
Os resultados mostram que a complexidade dos *Configurators* influencia nos seus tempos de resposta. O *Configurator* para o MRTG é mais complexo exigindo interações entre diversos componentes do MRTG, com o dispositivo a ser monitorado e com outros sistemas como *Cron* e Apache. Os *Configurators* para o Nagios criam os arquivos de configuração e informam o processo principal do Nagios para recuperar estas novas configurações. Esta diferença na complexidade dos *Configurators* imposta pelo método de configuração da solução de monitoramento fica evidente ao se analisar os tempos de resposta. Logo, deve-se desenvolver *Configurators* que utilizem métodos simples de configuração para alcançar melhores resultados com o FlexACMS. Entretanto, *Configurators* com tempo de resposta na ordem de 1 segundo, como o do MRTG, podem ser considerados adequados, visto que as demais operações envolvidas, como a criação de *cloud slices*, estão na ordem de uma dezena de segundos como mostram as Figuras 5.3a, 5.3b e 5.3c.

5.6 Tempo de resposta da arquitetura estendida

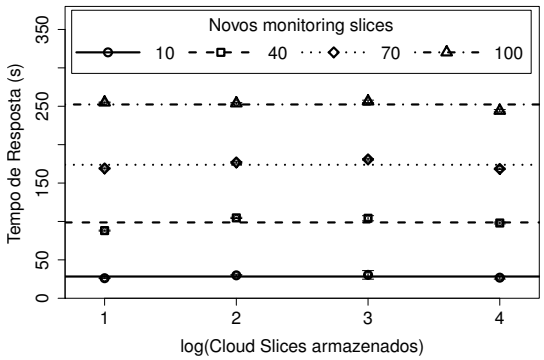
O objetivo deste experimento é analisar aspectos de escalabilidade em relação ao tempo de resposta da arquitetura estendida do FlexACMS variando-se fatores como o número de métricas por *monitoring slice*, o número de *monitoring slices* que devem ser configurados em uma rajada e o número de *cloud slices* armazenados no *framework*. O número de métricas em cada *monitoring slice* foi variado em 5, 25 e 50 métricas por *monitoring slice*, o que representa 1, 5 e 10 métricas por recurso do *cloud slice*. Considera-se que 50 métricas por *monitoring slice* (10 métricas por recurso) seja um grande número de métricas, pois o próximo nível para este fator parece exagerado porque utilizaria 15 métricas por recurso e um total de 75 métricas por *monitoring slice*. O número de *monitoring slices* que devem ser criados em rajada foi variado em 10, 40, 70 e 100 novos *monitoring slices* por rajada. Estima-se que a criação de 100 novos *monitoring slices* a partir de uma única requisição de um *Gatherer* seja um grande número de novos *cloud slices* criados na plataforma. Isto fica mais evidente quando se considera que requisições de *Gatherers* para o *Framework Core* serão enviadas através de um processo de *polling* de poucos minutos. O número de *cloud slices* armazenados no *framework* foi variado em 10, 100, 1000 e 10000 *cloud slices*. Até onde se conhece, não existem estatísticas disponíveis sobre o tamanho de cenários de provedores de nuvens *IaaS* reais. Estima-se que um provedor que hospeda 10000 *cloud slices* seja grande.

O cenário de avaliação foi composto por dois servidores: um servidor para o monitoramento e outro para o FlexACMS. Ambos possuem um processador Intel(R) Core(TM) i5 CPU 650@3.20GHz com 2 *cores*, 4GB de memória RAM, executando os sistemas operacionais Gentoo Linux e Ubuntu Server 12.10, respectivamente. Ambos foram conectados a um switch com enlaces de 1 Gbps de capacidade. O servidor de monitoramento executa 10 *Configurator Workers* simultaneamente e hospeda *Configurators* para o Nagios. O servidor do FlexACMS hospeda o *Framework Core*, um *Gatherer* que gera entradas artificiais e executa 10 trabalhadores que podem consumir simultaneamente tarefas das *Requests Queue* e *Changes Queue*. Foi utilizado o *Gatherer* eficiente, descrito na Se-

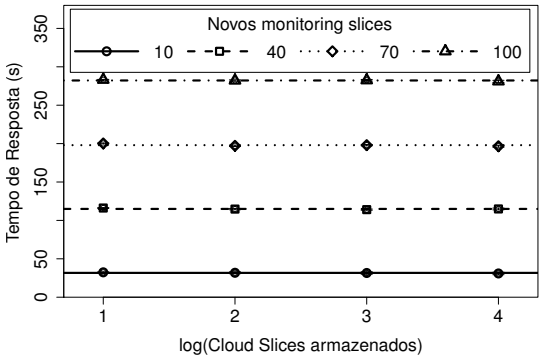
ção 5.1, que gera entradas artificiais sobre *cloud slices* com 20 informações e 5 recursos associados, que são valores semelhantes aos obtidos quando se utiliza o *Gatherer* desenvolvido para a OpenStack API apresentado na Seção 4.2. Cada observação foi repetida 30 vezes e os gráficos mostram intervalos com 95% de nível de confiança.



(a) 5 métricas por *monitoring slice*



(b) 25 métricas por *monitoring slice*



(c) 50 métricas por *monitoring slice*

Figura 5.6: Tempo de resposta da arquitetura estendida para avaliação de escalabilidade

As Figuras 5.6a, 5.6b e 5.6c mostram os resultados obtidos para *monitoring slices* com 5, 25 e 50 métricas, respectivamente. Através da análise dos gráficos pode-se verificar o efeito do aumento do número de métricas em cada *monitoring slice* comparando as linhas do mesmo padrão entre figuras vizinhas. Por exemplo, pode-se comparar as linhas das Figuras 5.6a e 5.6b que mostram resultados para 5 e 25 métricas, ou seja, *monitoring slices* que são 5 vezes maiores. O tempo de resposta na Figura 5.6b é um pouco maior do que na Figura 5.6a e a diferença é menor que 5 vezes. Pode-se obter conclusões semelhantes quando analisa-se as diferenças entre as Figuras 5.6a e 5.6c, que mostram resultados para 5 e 50 métricas, ou seja, *monitoring slices* que são 10 vezes maiores. Também pode-se chegar a estas conclusões analisando-se as Figuras 5.6b e 5.6c que mostram resultados para 25 e 50 métricas, ou seja, *monitoring slices* que são 2 vezes maiores. Portanto, pode-se concluir que o tempo de resposta do FlexACMS é adequado quando um número crescente de métricas são utilizadas em *monitoring slices*.

Através da análise dos gráficos das Figuras 5.6a, 5.6b e 5.6c pode-se observar o efeito do aumento do número de *monitoring slices* que precisam ser construídos em uma rajada.

Esta observação pode ser realizada ao se comparar a distância entre linhas adjacentes dentro de uma mesma figura. O número de *monitoring slices* que devem ser construídos em uma rajada foi variado em 10, 40, 70 e 100 *monitoring slices* por rajada, ou seja, foram variados por um número fixo de 30 *monitoring slices* por rajada. Portanto, o tempo de resposta também deve variar por um valor fixo entre as linhas da mesma figura. Pode-se observar que as linhas de uma mesma figura mantêm uma distância semelhante conforme aumenta o número de *monitoring slices* na rajada.

É difícil de medir as distâncias entre as linhas apenas observando os gráficos. Por isso, foram calculadas as taxas de vazão para cada cenário para verificar se o FlexACMS escala quando o número de *monitoring slices* em uma rajada aumenta. Para calcular a vazão, precisa-se do número total de métricas configuradas para cada linha, que pode ser calculado multiplicando-se o número de novos *monitoring slices* pelo número de métricas por *monitoring slice*. Por exemplo, 100 novos *monitoring slices* $*$ 50 métricas por *monitoring slice* $=$ 5000 métricas a serem configuradas. Ao dividir-se o número total de métricas pelo tempo de resposta se obtém a vazão em métricas configuradas por segundo. As linhas em cada figura apresentam vazão semelhantes que são em torno de 2,5 métricas, 10 métricas e 17 métricas configuradas por segundo nas Figuras 5.6a, 5.6b e 5.6c, respectivamente. Portanto, pode-se concluir que o tempo de resposta do FlexACMS é adequado quando aumenta-se o número de *monitoring slices* que precisam ser construídos em rajada.

Através da análise dos gráficos das Figuras 5.6a, 5.6b e 5.6c pode-se observar o efeito do aumento de *cloud slices* que estão armazenados no *framework*. Pode-se realizar esta observação acompanhando os pontos ao longo do eixo-x em cada figura. O eixo-x utiliza uma escala logarítmica para permitir uma comparação apropriada. O número de *cloud slices* foi variado de 10 até 10000 *cloud slices*, *i.e.*, de 10^1 até 10^4. Pode-se observar que os pontos acompanham paralelamente o eixo-x. Portanto, pode-se afirmar que o FlexACMS é escalável considerando o limite de 10000 *cloud slices* observados neste experimento.

Estes resultados foram analisados estatisticamente para avaliar a escalabilidade da arquitetura estendida do *framework* em relação ao aumento do número de *cloud slices* armazenados. Desta forma, foram realizadas regressões para modelos lineares e não-lineares para verificar a complexidade da solução em relação ao número de *cloud slices* armazenados (n). Foram verificadas as seguintes complexidades: $O(n)$, $O(n^2)$ e $O(e^n)$ para cada uma das curvas apresentadas nas Figuras 5.6a, 5.6b e 5.6c. A análise para $O(n^3)$ ficou prejudicada pelo número reduzido de pontos em cada curva (4 pontos), que é insuficiente para que a regressão seja realizada. Para estas complexidades, foram utilizados os mesmos modelos descritos na Tabela 5.1.

A Tabela 5.3 mostra as médias para os parâmetros estatísticos dos modelos obtidos pelas regressões para as curvas de uma mesma figura. Estes parâmetros, bem como as regressões, foram obtidos utilizando a função `lm` do software R (R Core Team, 2014). O parâmetro R^2 indica o quanto da variabilidade dos resultados consegue ser explicada pelo modelo obtido pela regressão. Pode-se verificar que o modelo para $O(n^2)$ apresenta os melhores valores de R^2 para os três casos, porém estes valores indicam que os modelos explicam no máximo 90% da variabilidade dos resultados. Além disso, o parâmetro *p-value* que indica a probabilidade de que o termo de maior ordem seja nulo é bastante elevada. Os modelos para $O(n)$ e $O(e^n)$, apesar de apresentarem bons valores para o caso de 5 métricas, apresentam valores inadequados nos demais casos como observado para o modelo de $O(n^2)$. Ao se observar as Figuras 5.6a, 5.6b e 5.6c pode-se perceber que o tempo de resposta não varia em relação à variação do eixo-x. Isto explica porque as regressões, de maneira geral, não oferecem valores adequados para os parâmetros estatís-

ticos R^2 e *p-value*. O que evidencia que o tempo de resposta não depende do número de *cloud slices* armazenados no *framework*, ou seja, o tempo de resposta não é afetado pelo número de *cloud slices* hospedados no ambiente de computação em nuvem.

Tabela 5.3: Média aritmética dos parâmetros estatísticos das regressões lineares e não-lineares dos tempos de resposta da arquitetura estendida para a construção de *monitoring slices* com 5, 25 e 50 métricas

Complexidade	**5 métricas**		**25 métricas**		**50 métricas**	
	R^2	*p-value*	R^2	*p-value*	R^2	*p-value*
$O(n)$	0,89516	0,05805	0,34215	0,52340	0,47356	0,38950
$O(n^2)$	0,90408	0,52730	0,66355	0,43617	0,68765	0,65776
$O(e^n)$	0,89718	0,05690	0,34189	0,52575	0,47585	0,38825

Conclui-se então que o tempo de resposta não é afetado pelo número de *cloud slices* hospedados no ambiente de computação em nuvem, ou seja, a complexidade da arquitetura estendida do *framework* é independente de n ($O(1)$). As linhas ilustradas nas Figuras 5.6a, 5.6b e 5.6c representam as médias dos tempos de resposta para rajadas de mesmo tamanho. Pode-se perceber que estas linhas representam com boa precisão os resultados obtidos. Pode-se concluir então que a arquitetura estendida alcança escalabilidade no tempo de resposta em relação ao número de *cloud slices* hospedados na plataforma, ou seja, em relação ao tamanho do ambiente de computação em nuvem, considerando o limite de 10000 *cloud slices* observados.

No decorrer deste capítulo foram discutidos os experimentos realizados para avaliar as arquiteturas inicial e estendida e seus componentes. A utilização de CPU e memória da arquitetura inicial mostrou-se adequada, porém também demonstrou que ela não utiliza plenamente os recursos disponíveis. Este aspecto prejudica a escalabilidade em relação ao tempo de resposta da arquitetura inicial, que varia quadraticamente em relação ao número de *cloud slices* no ambiente de computação em nuvem. Estas observações ensejaram o desenvolvimento da arquitetura estendida que melhora o desempenho da solução e permite a adição de novas funcionalidades ao *framework* como a atribuição automática e balanceada de tarefas de configuração. A arquitetura estendida é escalável por possuir tempo de resposta que independe do número de *cloud slices* no ambiente de computação em nuvem, considerando o limite de 10000 *cloud slices* observados. Neste sentido, sugere-se a arquitetura estendida como solução para a configuração automática de *monitoring slices* que utilizam múltiplas soluções de monitoramento.

6 CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação foi proposto um *framework* chamado FlexACMS para a construção automatizada de *monitoring slices* baseados em múltiplas soluções de monitoramento. Atualmente, são necessárias múltiplas soluções de monitoramento porque não existem soluções para computação em nuvem que sejam capazes de atender a todos os requisitos e funcionalidades exigidas pelos administradores de nuvem. Além disso, é inviável realizar manualmente a configuração destas soluções, principalmente daquelas que não são integradas às plataformas. Logo, esta tarefa deve ser apoiada por soluções de automatização que sejam integradas às plataformas e disparem a configuração dos *monitoring slices* quando *cloud slices* são criados ou modificados. O FlexACMS possibilita que administradores de nuvem configurem o monitoramento de seus *cloud slices* de maneira automatizada e flexível, visto que as métricas e soluções de monitoramento que serão utilizadas para monitorá-las são definidas através de regras pelos administradores.

O FlexACMS é uma solução independente das plataformas de computação em nuvem e das soluções de monitoramento. Para isso, utiliza-se componentes que atuam como *drivers* para comunicar-se com plataformas e soluções de monitoramento. Os *Gatherers* são responsáveis por obter informações sobre os *cloud slices* hospedados na plataforma de computação em nuvem e enviá-las para o *Framework Core*. O *Gatherer* utiliza APIs disponibilizadas pelas plataformas, como a OpenStack API disponibilizada pela plataforma OpenStack, para obter estas informações. O *Framework Core* processa estas informações em busca de operações realizadas na plataforma que exijam a configuração do monitoramento, como a criação de novos *cloud slices*. Com base nestas operações e em regras predefinidas pelos administradores da nuvem, são disparados *Configurators* que são os componentes responsáveis por configurar as soluções de monitoramento.

Duas arquiteturas foram propostas e avaliadas para o FlexACMS que foram chamadas de inicial e estendida. A arquitetura inicial utiliza uma abordagem serial para tratar as etapas de configuração dos *monitoring slices*, desde o processamento das informações enviadas pelos *Gatherers* até a configuração das soluções de monitoramento pelos *Configurators*. Esta abordagem pode ser comparada à abordagem adotada por administradores ao encadear uma série de *scripts* para realizar uma tarefa de configuração. A arquitetura estendida divide as etapas de configuração dos *monitoring slices* em tarefas menores que são processadas por trabalhadores especializados. Desta forma, diversas tarefas são executadas em paralelo para processar um conjunto de informações enviadas por um *Gatherer*. Esta característica permite a introdução de novas funcionalidades ao *framework* como a distribuição automatizada e balanceada de tarefas de configuração. As tarefas de configuração são colocadas em filas que são consumidas por trabalhadores instalados nos servidores de monitoramento, onde a distribuição ocorre a medida que os servidores concorrem pelas tarefas destas filas. O balanceamento é alcançado porque um servidor de

monitoramento pode decidir parar de consumir tarefas enquanto estiver sobrecarregado, deixando as novas tarefas para servidores que possuírem recursos disponíveis.

A arquitetura inicial foi avaliada em relação ao tempo de resposta e consumo de processamento e memória. O consumo de processamento e memória mostrou-se adequado. Entretanto, demonstrou-se que a arquitetura inicial não é capaz de explorar plenamente os recursos disponíveis nos servidores. Como o processamento da requisição é realizado de maneira serial, a arquitetura inicial utiliza apenas uma *thread* da CPU independentemente de quantas *threads* o servidor consegue processar simultaneamente. Os resultados obtidos para o tempo de resposta foram analisados através de modelos obtidos por regressões lineares e não-lineares para determinar a complexidade introduzida pela arquitetura. O melhor modelo obtido foi o quadrático ($O(n^2)$), ou seja, o tempo de resposta varia quadraticamente em relação ao número de *cloud slices* armazenados no *framework*. Estas duas observações levaram ao desenvolvimento da arquitetura estendida para contornar as restrições em relação ao desempenho da arquitetura inicial.

Foi realizado um comparativo entre os tempos de resposta das arquiteturas inicial e estendida para avaliar se as mudanças introduzidas melhoram o desempenho do *framework*. A arquitetura estendida reduz em cerca de 10% a influência do FlexACMS no tempo total do experimento que consiste em criar *cloud slices* no OpenStack e configurar automaticamente os *monitoring slices* correspondentes. Em termos absolutos, o tempo de resposta da arquitetura estendida é até 60% menor que o tempo de resposta da arquitetura inicial. Para o caso onde são construídos 10 *monitoring slices*, o tempo de resposta é reduzido de 49,76s para 20,2s. Logo, fica evidente que a arquitetura estendida possui desempenho melhor e que as mudanças introduzidas alcançaram seus objetivos.

O *Gatherer* desenvolvido para a OpenStack API também foi analisado em relação ao consumo de banda de comunicação para transmitir as informações da plataforma OpenStack para o *Framework Core*. Verificou-se que para cada *cloud slice* hospedado no OpenStack, o *Gatherer* utiliza em torno de 4 Kbytes para codificá-lo em XML para enviar suas informações para o REST *Web service* no *Framework Core*. Este consumo de banda de comunicação é adequado para utilização deste *Gatherer* em redes locais. Uma alternativa para reduzir o tamanho destas requisições seria a utilização de *Web services* baseados em padrões mais simples de codificação em substituição ao XML, como o JSON.

Os *Configurators* para Nagios e MRTG foram analisados em relação ao seu tempo de resposta. Foi verificado que a complexidade do método de configuração utilizado pela solução de monitoramento possui grande influência no tempo de resposta dos *Configurators*. O MRTG utiliza um método de configuração baseado em *scripts* que comunicam-se com o objeto a ser monitorado para obter informações para a criação de arquivos de configuração. Este método é mais complexo do que o utilizado pelo Nagios, que utiliza arquivos de configuração simples, cujas informações são recebidas do *Framework Core* como parâmetros pelos *Configurators*. O tempo de resposta do *Configurator* para o MRTG é até 24 vezes maior do que o tempo de resposta dos *Configurators* para o Nagios. Portanto, deve-se desenvolver *Configurators* que utilizem o método de configuração disponível que seja o mais simples possível para obter melhores tempos de resposta.

O tempo de resposta da arquitetura estendida também foi analisado em relação ao número de métricas por *monitoring slice*, número de *monitoring slices* construídos em uma rajada e número de *cloud slices* armazenados no *framework*. A análise dos gráficos para o tempo de resposta mostrou a escalabilidade da arquitetura estendida em relação a estes fatores, considerando o limite de 10000 *cloud slices* observados. Em relação ao número de *cloud slices* armazenados no *framework*, foi realizada uma análise através de

modelos obtidos por regressões lineares e não-lineares para determinar a complexidade da arquitetura. Os modelos obtidos para $O(n)$, $O(n^2)$ e $O(e^n)$ não atingiram bons resultados e ao visualizar os gráficos, pode-se perceber que o tempo de resposta é independente do número de *cloud slices* armazenados. Logo, a arquitetura estendida é escalável em relação ao número de *cloud slices* armazenados, ou seja, em relação ao tamanho do ambiente que está sendo monitorado, considerando o limite de 10000 *cloud slices* observados.

A arquitetura estendida do FlexACMS consome mais recursos computacionais que a arquitetura inicial, pois explora o paralelismo disponível no hardware. Porém, a arquitetura estendida permite que o consumo de recursos seja ajustado pelo administrador ao estabelecer o número de trabalhadores que serão executados paralelamente. Cabe salientar que as avaliações do FlexACMS foram realizadas em ambientes com pouca capacidade de processamento compostos por computadores pessoais. Além disso, os cenários de avaliação foram baseados em um servidor único para hospedar o *Framework Core*. Mesmo assim, pode-se concluir que os tempos de resposta alcançados com estes ambientes são apropriados. No entanto, existem alternativas para aumentar o desempenho do FlexACMS, caso seja necessário. A arquitetura estendida permite que sejam utilizados diversos servidores para hospedar componentes como trabalhadores que realizam o processamento mais oneroso do *framework*. Adicionalmente, as tecnologias utilizadas pelo *framework* como servidores *Web* Apache, banco de dados MySQL e sistema de filas Redis possuem mecanismos para aumentar seu desempenho, tais como balanceadores de carga.

Os objetivos propostos para esta dissertação foram alcançados, pois o FlexACMS permite que administradores de ambientes de computação em nuvem utilizem as soluções de monitoramento que atendam as suas necessidades independentemente da integração existente entre solução e plataforma de computação em nuvem. O FlexACMS também automatiza as tarefas de configuração das soluções de acordo com as operações realizadas na plataforma utilizando regras predefinidas pelos administradores do ambiente de computação em nuvem. Além disso, a avaliação de desempenho demonstrou que o FlexACMS possui tempo de resposta adequado e que a arquitetura estendida é escalável em relação ao número de *cloud slices* do ambiente, considerando os 10000 *cloud slices* observados. Portanto, sugere-se a arquitetura estendida como solução para a criação automatizada de *monitoring slices* baseados em múltiplas soluções de monitoramento.

Como trabalhos futuros, pode-se completar o ciclo de vida de um *cloud slice*, visto que a versão atual do FlexACMS apenas realiza a criação de *monitoring slices* para reagir à criação de *cloud slices* na plataforma. Para tanto, é necessário completar o desenvolvimento do protótipo para reagir a reconfigurações e deleções de *cloud slices* na plataforma de computação em nuvem realizando o ajuste e a deleção dos *monitoring slices* quando necessário. Além disso, como trabalho futuro, pode-se investigar a viabilidade do FlexACMS para outros tipos de nuvens como PaaS e SaaS.

Por fim, esta dissertação provou que a construção automatizada de *monitoring slices* é viável e factível. Portanto, a partir deste trabalho espera-se que soluções de monitoramento e plataformas de computação em nuvem evoluam para contemplar as funcionalidades fornecidas pelo FlexACMS. As soluções de monitoramento podem oferecer integração com plataformas de nuvem para reagir à criação de *cloud slices* de acordo com regras predefinidas pelos administradores. Por outro lado, as plataformas de computação em nuvem podem oferecer mecanismos para configurar *monitoring slices* a partir da criação de *cloud slices* também baseando-se em regras predefinidas pelos administradores. Desta forma, o monitoramento de ambientes de computação em nuvem se tornará tão flexível quanto o monitoramento de ambientes tradicionais.

REFERÊNCIAS

ACETO, G. et al. Cloud Monitoring: A Survey. **Computer Networks**, [S.l.], v.57, n.9, p.2093 – 2115, 2013.

ARMBRUST, M. et al. **Above the Clouds**: A Berkeley View of Cloud Computing. [S.l.: s.n.], 2009.

ARMBRUST, M. et al. A View of Cloud Computing. **Communications of the ACM**, New York, NY, USA, v.53, n.4, p.50–58, Apr. 2010.

ASSUNCAO, M. D. de; COSTANZO, A. di; BUYYA, R. Evaluating the Cost-benefit of Using Cloud Computing to Extend the Capacity of Clusters. In: INTERNATIONAL ACM SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 18., New York, NY, USA. **Proceedings...** ACM, 2009. p.141–150. (HPDC '09).

BARHAM, P. et al. Xen and the Art of Virtualization. **ACM SIGOPS Operating Systems Review**, New York, NY, USA, v.37, n.5, p.164–177, Oct. 2003.

BARI, M. F. et al. Data Center Network Virtualization: A Survey. **IEEE Communications Surveys & Tutorials**, [S.l.], v.15, n.2, p.909–928, 2013.

BASET, S. A. Cloud SLAs: Present and Future. **ACM SIGOPS Operating Systems Review**, New York, NY, USA, v.46, n.2, p.57–66, July 2012.

BOLTE, M. et al. Non-intrusive Virtualization Management Using libvirt. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE. **Proceedings...** [S.l.: s.n.], 2010. p.574–579.

BONIFACE, M. et al. Platform-as-a-Service Architecture for Real-Time Quality of Service Management in Clouds. In: INTERNATIONAL CONFERENCE ON INTERNET AND WEB APPLICATIONS AND SERVICES (ICIW), 15. **Proceedings...** [S.l.: s.n.], 2010. p.155–160.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. **Pattern-oriented Software Architecture**: On Patterns and Pattern Language. [S.l.]: John Wiley & Sons, 2007. v.5.

BUYYA, R. et al. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. **Future Generation Computer Systems**, Amsterdam, The Netherlands, v.25, n.6, p.599–616, June 2009.

BUYYA, R.; YEO, C. S.; VENUGOPAL, S. Market-oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS (HPCC '08), 10. **Proceedings...** [S.l.: s.n.], 2008. p.5–13.

CALERO, J. A.; AGUADO, J. G. MonPaaS: An Adaptive Monitoring Platform as a Service for Cloud Computing Infrastructures and Services. **IEEE Transactions on Services Computing**, [S.l.], v.8, n.1, p.65–78, 2015.

CARLSON, J. L. **Redis in Action**. Greenwich, CT, USA: Manning Publications Co., 2013.

CARVALHO, M. et al. A Cloud Monitoring Framework for Self-configured Monitoring Slices Based on Multiple Tools. In: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT 2013 (CNSM 2013), 9. **Proceedings...** [S.l.: s.n.], 2013.

CARVALHO, M. et al. Efficient Configuration of Monitoring Slices for Cloud Platform Administrators. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (IEEE ISCC 2014), 19. **Proceedings...** [S.l.: s.n.], 2014.

CARVALHO, M.; GRANVILLE, L. Z. Incorporating Virtualization Awareness in Service Monitoring Systems. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT (IM 2011), 12. **Proceedings...** [S.l.: s.n.], 2011. p.297–304.

CLAYMAN, S. et al. Monitoring Service Clouds in the Future Internet. In: FUTURE INTERNET ASSEMBLY. **Proceedings...** IOS Press, 2010. p.115–126.

CLAYMAN, S. et al. Monitoring, Aggregation and Filtering for Efficient Management of Virtual Networks. In: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT (CNSM 2011), 7. **Proceedings...** [S.l.: s.n.], 2011. p.1–7.

CLAYMAN, S.; GALIS, A.; MAMATAS, L. Monitoring Virtual Networks with Lattice. In: IEEE/IFIP NETWORK OPERATIONS AND MANAGEMENT SYMPOSIUM WORKSHOPS (NOMS WORKSHOPS). **Proceedings...** [S.l.: s.n.], 2010. p.239–246.

CORRADI, A. et al. DDS-enabled Cloud Management Support for Fast Task Offloading. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (IEEE ISCC 2012), 17. **Proceedings...** [S.l.: s.n.], 2012. p.67–74.

DE CHAVES, S.; URIARTE, R.; WESTPHALL, C. Toward an Architecture for Monitoring Private Clouds. **IEEE Communications Magazine**, [S.l.], v.49, n.12, p.130 –137, Dec. 2011.

DIKE, J. **User Mode Linux**. [S.l.]: Prentice Hall Englewood Cliffs, 2006. v.2.

ELMROTH, E. et al. Accounting and Billing for Federated Cloud Infrastructures. In: INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING (GCC'09), 8. **Proceedings...** [S.l.: s.n.], 2009. p.268–275.

ERL, T. **SOA**: Principles of Service Design. [S.l.]: Prentice Hall Upper Saddle River, 2008. v.1.

ESTEVES, R. P. et al. Paradigm-based Adaptive Provisioning in Virtualized Data Centers. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT (IM 2013), 13. **Proceedings...** [S.l.: s.n.], 2013. p.169–176.

FATEMA, K. et al. A Survey of Cloud Monitoring Tools: Taxonomy, Capabilities and Objectives. **Journal of Parallel and Distributed Computing**, [S.l.], v.74, n.10, p.2918 – 2933, 2014.

FIFIELD, T. et al. **OpenStack Operations Guide**. [S.l.]: O'Reilly Media, Inc., 2014.

FLANAGAN, D.; MATSUMOTO, Y. **The Ruby Programming Language**. [S.l.]: O'Reilly Media, Inc., 2008.

FOSTER, I. et al. Cloud Computing and Grid Computing 360-degree Compared. In: GRID COMPUTING ENVIRONMENTS WORKSHOP (GCE'08), 4. **Proceedings...** [S.l.: s.n.], 2008. p.1–10.

FRANCESCHI, A. **Extending Puppet**. [S.l.]: Packt Publishing Ltd, 2014.

GOUDARZI, H.; GHASEMAZAR, M.; PEDRAM, M. SLA-based Optimization of Power and Migration Cost in Cloud Computing. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING (CCGRID 2012), 2012., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2012. p.172–179.

HARRINGTON, D.; PRESUHN, R.; WIJNEN, B. **An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks**. [S.l.]: IETF, 2002. n.3411. (Request for Comments).

HARSH, P. et al. Using Open Standards for Interoperability Issues, Solutions, and Challenges Facing Cloud Computing. In: INTERNATIONAL CONFERENCE ON NETWORK AND SERVICE MANAGEMENT (CNSM 2012) AND 6TH WORKSHOP ON SYSTEMS VIRTUALIZATION MANAGEMENT (SVM 2012), 8. **Proceedings...** [S.l.: s.n.], 2012. p.435–440.

HARTL, M.; FERNANDEZ, O. **The Ruby on Rails 3 Tutorial and Reference Collection**. [S.l.]: Addison-Wesley Professional, 2011.

JAIN, R. **The Art of Computer Systems Performance Analysis**: Techniques for Experimental Design, Measurement, Simulation, and Modeling. [S.l.: s.n.], 1991. xxvii + 685p.

JOSEPHSEN, D. **Nagios**: Building Enterprise-grade Monitoring Infrastructures for Systems and Networks. [S.l.]: Prentice Hall Press, 2013.

KIM, H.; FEAMSTER, N. Improving Network Management with Software Defined Networking. **IEEE Communications Magazine**, [S.l.], v.51, n.2, p.114–119, Feb. 2013.

KIVITY, A. et al. KVM: The Linux Virtual Machine Monitor. In: LINUX SYMPOSIUM. **Proceedings...** [S.l.: s.n.], 2007. v.1, p.225–230.

KNUDSEN, J.; NIEMEYER, P. **Learning Java**. [S.l.]: O'Reilly, 2005.

KOCJAN, W. **Learning Nagios 4**. [S.l.]: Packt Publishing Ltd, 2014.

KOSLOVSKI, G. et al. Executing Distributed Applications on Virtualized Infrastructures Specified with the VXDL Language and Managed by the HIPerNET Framework. In: **Cloud Computing**. [S.l.]: Springer, 2010. p.3–19.

KUTARE, M. et al. Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers. In: INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING, 7. **Proceedings...** [S.l.: s.n.], 2010. p.141–150. (ICAC '10).

LAURENT, S. S. et al. **Programming Web Services with XML-RPC**. [S.l.]: O'Reilly Media, Inc., 2001.

LAURIE, B.; LAURIE, P. **Apache**: The Definitive Guide. [S.l.]: O'Reilly Media, Inc., 2003.

LERDORF, R.; TATROE, K.; MACINTYRE, P. **Programming PHP**. [S.l.]: O'Reilly Media, Inc., 2006.

LONEA, A. Private Cloud Set Up Using Eucalyptus Open Source. In: BALAS, V. E. et al. (Ed.). **Soft Computing Applications**. [S.l.]: Springer Berlin Heidelberg, 2013. p.381–389. (Advances in Intelligent Systems and Computing, v.195).

LOWE, S. **Mastering VMware vSphere 5**. [S.l.]: John Wiley & Sons, 2011.

LU, M.; CHIUEH, T. Fast Memory State Synchronization for Virtualization-based Fault Tolerance. In: IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS NETWORKS (DSN'09), 39. **Proceedings...** [S.l.: s.n.], 2009. p.534–543.

LUTZ, M. **Learning Python**. [S.l.]: O'Reilly Media, Inc., 2013.

MARSCHALL, M. **Chef Infrastructure Automation Cookbook**. [S.l.]: Packt Publishing Ltd, 2013.

MELL, P.; GRANCE, T. The NIST Definition of Cloud Computing (Draft). **NIST Special Publication**, [S.l.], v.800, p.145, 2011.

MENG, S.; LIU, L. Enhanced Monitoring-as-a-Service for Effective Cloud Management. **IEEE Transactions on Computers**, [S.l.], v.62, n.9, p.1705–1720, 2013.

MONTES, J. et al. GMonE: A Complete Approach to Cloud Monitoring. **Future Generation Computer Systems**, [S.l.], p.–, 2013.

MURTY, J. **Programming Amazon Web Services**: S3, EC2, SQS, FPS, and SimpleDB. [S.l.]: O'Reilly Media, Inc., 2008.

NEWHAM, C.; ROSENBLATT, B. **Learning the Bash Shell**: Unix Shell Programming. [S.l.]: O'Reilly Media, Inc., 2005.

NURMI, D. et al. The Eucalyptus Open-source Cloud-computing System. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID '09), 9. **Proceedings...** [S.l.: s.n.], 2009. p.124–131.

OGLESBY, R.; HEROLD, S. VMware ESX Server: Advanced Technical Design Guide (Advanced Technical Design Guide Series). **The Brian Madden Company**, [S.l.], 2005.

OREBAUGH, A.; RAMIREZ, G.; BEALE, J. **Wireshark & Ethereal Network Protocol Analyzer Toolkit**. [S.l.]: Syngress, 2006.

PARDO-CASTELLOTE, G.; FARABAUGH, B.; WARREN, R. An Introduction to DDS and Data-centric Communications. **Real-Time Innovations. OpenURL**, [S.l.], 2005.

PEPPLE, K. **Deploying OpenStack**. [S.l.]: O'Reilly Media, 2011.

POSTEL, J. **Internet Control Message Protocol**. [S.l.]: IETF, 1981. n.792. (Request for Comments).

R Core Team. **R**: A Language and Environment for Statistical Computing. Vienna, Austria: R Foundation for Statistical Computing, 2014.

RAJSHEKHAR, A. **.Net Framework 4.5 Expert Programming Cookbook**. [S.l.]: Packt Publishing Ltd, 2013.

RAK, M. et al. Cloud Application Monitoring: The mOSAIC Approach. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING TECHNOLOGY AND SCIENCE (CLOUDCOM 2011), 3. **Proceedings...** [S.l.: s.n.], 2011. p.758 –763.

RHOTON, J. **Cloud Computing Explained**. [S.l.]: Recursive Limited, 2009.

RICHARDSON, L.; RUBY, S. **RESTful Web Services**. [S.l.]: O'Reilly Media, Inc., 2008.

RODRIGUES, G. et al. An Architecture to Evaluate Scalability, Adaptability and Accuracy in Cloud Monitoring Systems. In: INTERNATIONAL CONFERENCE ON INFORMATION NETWORKING (ICOIN 2014), 28. **Proceedings...** [S.l.: s.n.], 2014. p.46–51.

SCHWARTZ, R. L.; PHOENIX, T. **Learning Perl**. [S.l.]: O'Reilly & Associates, Inc., 2008.

SEFRAOUI, O.; AISSAOUI, M.; ELEULDJ, M. OpenStack: Toward an Open-source Solution for Cloud Computing. **International Journal of Computer Applications**, [S.l.], v.55, n.3, p.38–42, Oct. 2012.

SHAO, J. et al. A Runtime Model Based Monitoring Approach for Cloud. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, 3. **Proceedings...** [S.l.: s.n.], 2010. p.313–320.

SHIPWAY, S.; OETIKER, T. **Using MRTG with RRDtool and Routers2**. [S.l.]: SJ Shipway, 2010.

SIMÕES, J. M. B. **Monitorização Automática de Redes de Computadores**: Estudo e Proposta de uma Nova Solução. 2010. Dissertação de Mestrado — Universidade Nova de Lisboa.

SOTOMAYOR, B. et al. Virtual Infrastructure Management in Private and Hybrid Clouds. **IEEE Internet Computing**, [S.l.], v.13, n.5, p.14–22, 2009.

SRIPARASA, S. S. **JavaScript and JSON Essentials**. [S.l.]: Packt Publishing Ltd, 2013.

SUBASHINI, S.; KAVITHA, V. A Survey on Security Issues in Service Delivery Models of Cloud Computing. **Journal of Network and Computer Applications**, [S.l.], v.34, n.1, p.1 – 11, 2011.

TAKABI, H.; JOSHI, J. B.; AHN, G. Security and Privacy Challenges in Cloud Computing Environments. **IEEE Security & Privacy**, [S.l.], p.24–31, 2010.

TORALDO, G. **OpenNebula 3 Cloud Computing**. [S.l.]: Packt Publishing, 2012. (Community Experience Distilled).

TULLOCH, M. **Introducing Windows Azure for IT Professionals**. [S.l.]: Microsoft Press, 2013.

VACCHE, A. D.; LEE, S. K. **Mastering Zabbix**. [S.l.]: Packt Publishing Ltd, 2013.

VAN VLIET, J.; PAGANELLI, F. **Programming Amazon EC2**. [S.l.]: O'Reilly Media, Inc., 2011.

VAQUERO, L. M. et al. A Break in the Clouds: Towards a Cloud Definition. **ACM SIGCOMM Computer Communication Review**, New York, NY, USA, v.39, n.1, p.50–55, Dec. 2008.

VELTE, A.; VELTE, T. **Microsoft Virtualization with Hyper-V**. [S.l.]: McGraw-Hill, Inc., 2009.

VOLK, E. et al. Towards Intelligent Management of Very Large Computing Systems. In: BISCHOF, C. et al. (Ed.). **Competence in High Performance Computing 2010**. [S.l.]: Springer Berlin Heidelberg, 2012. p.191–204.

WANG, L. et al. Cloud Computing: A Perspective Study. **New Generation Computing**, [S.l.], v.28, n.2, p.137–146, 2010.

WEN, X. et al. Comparison of Open-source Cloud Management Platforms: OpenStack and OpenNebula. In: INTERNATIONAL CONFERENCE ON FUZZY SYSTEMS AND KNOWLEDGE DISCOVERY (FSKD 2012), 9. **Proceedings...** [S.l.: s.n.], 2012. p.2457–2461.

WICKBOLDT, J. A. et al. Rethinking Cloud Platforms: Network-aware Flexible Resource Allocation in IaaS Clouds. In: IFIP/IEEE INTERNATIONAL SYMPOSIUM ON INTEGRATED NETWORK MANAGEMENT (IM 2013), 13. **Proceedings...** [S.l.: s.n.], 2013. p.450–456.

WICKBOLDT, J. A. et al. Resource Management in IaaS Cloud Platforms Made Flexible through Programmability. **Computer Networks**, [S.l.], 2014.

YAZIR, Y. et al. Dynamic Resource Allocation in Computing Clouds Using Distributed Multiple Criteria Decision Analysis. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING (CLOUD 2010), 3. **Proceedings...** [S.l.: s.n.], 2010. p.91–98.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud Computing: State-of-the-art and Research Challenges. **Journal of Internet Services and Applications**, [S.l.], v.1, n.1, p.7–18, 2010.

APÊNDICE A - ARTIGO PUBLICADO - ISCC 2014

Este apêndice apresenta o artigo *"Efficient Configuration of Monitoring Slices for Cloud Platform Administrators"*. Este artigo é resultado do aprimoramento no desempenho e introdução de novas funcionalidades da arquitetura estendida do FlexACMS. Este artigo apresenta o comparativo do tempo de resposta entre a arquitetura inicial e estendida e os resultados da avaliação de escalabilidade da arquitetura estendida.

- Título: Efficient Configuration of Monitoring Slices for Cloud Platform Administrators
- Evento: 19^{th} IEEE Symposium on Computers and Communications (ISCC 2014)
- URL: http://www.ieee-iscc.org/
- Data: 23 a 26 de junho de 2014
- Local: Madeira, Portugal

Efficient Configuration of Monitoring Slices for Cloud Platform Administrators

Márcio Barbosa de Carvalho*, Rafael Pereira Esteves*, Guilherme da Cunha Rodrigues*,
Clarissa Cassales Marquezan†, Lisandro Zambenedetti Granville*, Liane Margarida Rockenbach Tarouco*
*Institute of Informatics – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
Email: {mbcarvalho, rpesteves, gcrodrigues, granville}@inf.ufrgs.br, liane@penta.ufrgs.br
†Paluno, University of Duisburg-Essen, Germany
Email: clarissa.marquezan@paluno.uni-due.de

***Abstract*—Monitoring is an important issue in cloud environments because it assures that acquired cloud slices attend the user's expectations. However, these environments are multi-tenant and dynamic, requiring automation techniques to offload cloud administrators. In a previous work, we proposed FlexACMS: a framework to automate monitoring configuration related to cloud slices using multiple monitoring solutions. In this work, we enhanced FlexACMS to allow dynamic and automatic attribution of monitoring configuration tasks to servers without administrator intervention, which was not available in previous version. FlexACMS also considers the monitoring server load when attributing configuration tasks, which allows load balancing between monitoring servers. The evaluation showed that enhancements reduced FlexACMS response time up to 60% in comparison to previous version. The scalability evaluation of enhanced version demonstrated the feasibility of our approach in large scale cloud environments.**

***Index Terms*—Cloud computing, monitoring configuration.**

I. Introduction

Infrastructure as a Service (IaaS) [1] is a cloud model that offers *cloud slices* to cloud users. These cloud slices are indeed virtualized computational resources (*e.g.*, compute, storage, network) that need to be closely monitored to guarantee the quality of service expected by IaaS cloud users. In addition to cloud slices, Infrastructure Providers (InPs) also offer Monitoring as a Service (MaaS) [2]. In this case, cloud users can also subscribe to personalized monitoring services associated with their cloud slices. InPs also need monitoring results to enable efficient physical resource utilization; to avoid wasting expensive resources that are over provisioned. The cloud monitoring therefore directly impacts the InP revenue: by keeping cloud user's subscription guarantying their expectations, earning extra revenue provided by MaaS, and enabling efficient resource utilization of the cloud infrastructure.

When a cloud user requests a new cloud slice, the monitoring support for this new cloud slice must be also configured. We define the set of monitoring support (*e.g.*, monitored metrics and required configuration) associated with one cloud slice as a *monitoring slice* [3]. Current monitoring solutions cannot satisfy all cloud platform administrator requirements because these solutions cannot support all the possible monitoring requirements that different cloud slices might have. This means that cloud slices need to be monitored by a set of monitoring solutions [4] instead of a single unified solution. In addition, some monitoring solutions required by cloud slices are not integrated with cloud platforms. As a consequence, cloud platform administrators need to manually configure solutions to monitor metrics of monitoring slices or develop specialized and individual scripts for this configuration.

In a previous work [3] we proposed a framework called Flexible Automated Cloud Monitoring Slices (FlexACMS) to address the problem of automatically setting up monitoring slices. Despite the increase in the automation this work introduced, it remained a task of the cloud platform administrator to manually configure which set of metrics from a monitoring slice would be attributed, *i.e.* mapped, to which monitoring server. For instance, CPU metrics of monitoring slice#1 are configured to be responsibility of monitoring server Nagios#A, while CPU metrics of monitoring slice#2 are responsibility of Nagios#B. As consequence, there was no automatic mechanism to better distribute the configuration load among the monitoring servers during the configuration task, *i.e.*, the process of creating the monitoring slices.

In this paper, we extended the previous version of the FlexACMS framework solving the two aforementioned drawbacks and enhanced its automation and performance on configuring monitoring slices. The three main contributions of this paper are: *(i)* the dynamic and automatic attribution of configuration tasks to monitoring servers in a Message-Queuing fashion; *(ii)* load balancing among monitoring servers when attributing configuration tasks; and *(iii)* exploring the intrinsic parallelism in building monitoring slices.

We provide an analysis of the overhead introduced by the FlexACMS in terms of response time during the process of automatically configuring the support for new cloud users of IaaS and monitoring services (*i.e.*, creation of a cloud slice and its required monitoring slice). In the evaluated IaaS scenario, OpenStack [5] was used as cloud platform to create cloud slices; while Nagios [6] and MRTG [7] are the monitoring solutions associated with the monitoring slice to be configured by FlexACMS. The results show that the enhanced version has little impact on the response time during the automatic configuration of IaaS and monitoring services of InPs. We also evaluated the scalability of FlexACMS in terms of response time when varying the number of: cloud slices in place,

monitoring slices to be created, and metrics per monitoring slice. This scalability evaluation demonstrated the feasibility of our approach for large scale cloud environments.

This paper is organized as follows. In Section 2, the state-of-the-art on cloud computing monitoring is reviewed. In Section 3, we review the previous version of FlexACMS framework and its limitations. In Section 4, we detail the new features and enhancements proposed for the FlexACMS framework. In Section 5, we comment the results of both functional and scalability evaluations. In Section 6, we discuss our conclusions and future work.

II. Related Work

A variety of monitoring solutions are available for gathering updated status of cloud slices and applications deployed in these cloud slices. For instance, PCMONS [8] is a monitoring system for IaaS cloud models that uses a layered architecture to handle heterogeneity in the infrastructure, which provides an abstraction that allows the monitoring integration of different cloud platforms. Global Monitoring systEm (GmonE) [4] is also a monitoring solution for IaaS cloud models which provides a complete approach to cloud monitoring including metrics to users and administrators. However, these solutions need to be improved to have the ability of monitor different type of resources (*e.g.*, web servers, databases, applications).

InPs and cloud platforms also offer monitoring solutions to their costumers, such as Amazon CloudWatch [9] and Ceilometer [10]. Amazon CloudWatch is the monitoring service for Amazon Web Services (AWS) [11] that offers predefined basic metrics and allows user-defined metrics that are charged apart in a MaaS manner. Ceilometer aims to be the monitoring infrastructure for OpenStack platform to collect and share monitoring data with external consumers. However, these solutions are specific to these InP services and platforms, which hamper service/platform choice and migration.

In addition, traditional monitoring solutions can also be employed in cloud computing monitoring, such as Nagios [6] and MRTG [7]. These solutions are usually employed in heterogeneous environments, which are similar to cloud computing. For instance, Nagios is used for monitoring of different type of resources (*e.g.*, web servers, databases, applications) that are also deployed in cloud computing. Furthermore, the administrator's know-how in these solutions facilitates their use and extensibility. However, traditional monitoring solutions are not integrated with cloud platforms to be configured automatically, which requires manual configuration, scripting techniques, or discovery process. The latter is an expensive approach, especially in cloud computing environments where new cloud slices arrive and leave in a dynamic way.

There are solutions that monitor all layers of a cloud application stack, such as Runtime Model for Cloud Monitoring (RMCM) [12] and GmonE [4]. Other examples of cloud monitoring solutions include: the mOSAIC framework [13] that allows the development of cloud monitoring systems through the mOSAIC API, and the RESTFul Cloud Management System (RESTFul CMS) [14] that includes a cloud monitoring system based on RESTful Web services.

Aceto *et al.* [15] provide a wide study about available cloud monitoring solutions. Beyond, they also analyzed the cloud monitoring properties and requirements that monitoring solutions need to provide. Through their study we can conclude that there is not a single solution that addresses all monitoring properties and requirements. Thus, we believe that integrating multiple monitoring solutions cannot be ignored as an approach to build cloud monitoring systems.

There are also solutions like Puppet [16] and Chef [17] that could be used to automate configuration in general. Their clients are installed on server/hosts that retrieve configuration from a centralized configuration database. In cloud monitoring configuration the inverse behavior is expected, when a cloud slice (host) is created, the monitoring servers should retrieve new configurations. Thus, these solutions must be adapted to be appropriated for cloud monitoring configuration.

III. Review on FlexACMS

In this section, we review the key concepts and elements of FlexACMS framework introduced in our previous work [3]. Then, we discuss the limitations that motivated the enhancements that we propose now in this paper. The most important concept introduced by our framework is the *Monitoring Slice*. It reflects all monitoring information about a cloud slice, which is composed by both metrics related to a cloud slice and by the corresponding monitoring configuration to perform the collection. Multiple monitoring solutions might be used to collect these metrics [4]. Examples of monitoring solutions that can be combined in the collection of information from a cloud slice are: Nagios [6], MRTG [7], and Ceilometer [10].

The previous version of the FlexACMS framework created the support for automating the process of configuring the monitoring slice. The elements of the previous version of framework are: *Gatherers*, *FlexACMS Core* and *Configurators*. The *Gatherers* are responsible for collecting information about cloud slices hosted in cloud platforms and for sending this information to the Framework Core element through the REST Web service. The *FlexACMS Core* is responsible for processing the information about cloud slices received from gatherers. It stores the received cloud platform information to enable the detection of changes (*e.g.*, cloud slice creation). Based on the detection of these changes, the *FlexACMS Core* triggers the *Configurators* to build the corresponding monitoring slice for new detected cloud slice. Internally the *FlexACMS Core* was composed of three modules: REST Web Service, that receives the information from *Gatherers*; Change detection, responsible for comparing the stored information of created monitoring slices with the information about the existent cloud slices; and the Configuration Executor, that was designed to trigger the *Configurator*, passing as parameters the information needed to set up the monitoring slice. Finally, the *Configurator* is responsible for conducting the process of setting up the monitoring slices by configuring the monitoring servers that will collect the information associated with such slice.

Previous FlexACMS version freed cloud administrators of configuring each one of the monitoring servers themselves to support the collection of information about each new cloud slice. However, the cloud administrator still had to statically configure the set of metrics to be monitored by each monitoring server, *e.g.*, CPU metrics of monitoring slice of type #1 will be monitored by server Nagios#A, and CPU metrics of monitoring slice type #2 will be monitored by Nagios #B.

In the previous version of FlexACMS, the attribution of configuration tasks was done statically, as discussed above. We define the term attribution of the configuration tasks to denote this process of defining which monitoring server will be responsible for receiving request for setting up the monitoring slice support for a given type of monitoring slice. Previous FlexACMS version also did not take into account that the load of monitoring servers can change dynamically. Thus, the manual and static attribution of configuration tasks could lead to unbalanced distribution of the configuration load among the monitoring servers. In turn, this leads to potential loses of performance in the monitoring support.

IV. Enhanced FlexACMS Framework

In this paper, we propose the enhanced FlexACMS framework that is able to tackle problems related to attribution of configuration tasks to monitoring servers. To tackle the aforementioned problems and leverage the automation of FlexACMS, we introduced two major features:

- Dynamic and automatic attribution of configuration tasks in a Message-Queueing fashion: enables the automatic mapping between type of metrics to be monitored in a monitoring slice and the type of monitoring server to consume the request for configuration of the metric. This means that there is not a static specific monitoring server that will consume a request for configuration of a monitoring slice. Now any monitoring server can register to receive requests of a certain type based on the attributes configured by the cloud administrator.
- Load balancing during the configuration of the monitoring slices: enables the monitoring servers to volunteer to receive a task (i.e., setting up the monitoring slice) when they have capacity for doing so. The decision of registering can be taken based on different criteria. In this paper, we consider the load of the monitoring server.

Our previous FlexACMS architecture [3] was single-threaded, thus the configuration of monitoring slices was handled as a large task and all required steps were performed one after another. This approach did not consume much computational resources, but it did not also explore parallelism in performing these tasks and the parallel architecture of current computers. We thus extended FlexACMS to break the task of configure monitoring slices into several small tasks.

A. New architecture

The enhanced FlexACMS framework has significant structural changes in its core elements. The new architecture is illustrated in Figure 1, where all gray elements denote the changes introduced in the new version of the framework. Along this section, we detail the queue and workers components introduced in enhanced FlexACMS to enable the new features. Other components such as Gatherers, REST Web Service, and Configurators were briefly presented in Section III, and they were not modified in the enhanced version.

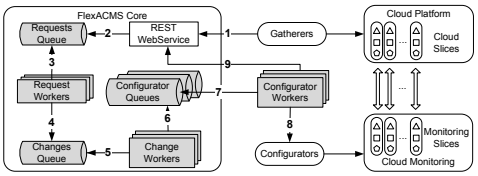


Fig. 1. FlexACMS architecture

FlexACMS Core receives information about cloud slices from gatherers through the REST Web Service (Figure 1, step 1). The FlexACMS Core performs tasks over the received information until triggering appropriate Configurators (steps 2-6). These tasks are processed by queue/workers to explore the intrinsic parallelism of these tasks. Firstly, the received information is queued in the *Requests Queue* (step 2) to be processed by a *Request Worker* (step 3). Request Workers are responsible for detecting changes between received information and previously stored information about the cloud slices deployed in the cloud platform. Each change detected is queued in the *Changes Queue* (step 4) to be processed by a *Change Worker* (step 5). Change Workers are responsible for evaluate whether detected change satisfies all Configurator's interests and conditions (see Table I), which are rules predefined by the cloud administrator. This means that these rules are checked to determine whether a new monitoring slice should be created for the new cloud slice. When all these rules are satisfied, the Change Worker queues a configurator call in the appropriated *Configurator Queue* (step 6) indicated by the cloud administrator. The configurator call has an identification and the command that executes the Configurator within all parameters necessary to build the monitoring configuration.

Configurator Queues represent pools of monitoring servers that share some characteristics. Table I shows these characteristics, which we call configurator attributes, and they include the queue where the configurator calls will be placed. In this example, we have two Configurator Queues: nagios_basic and nagios_platinum, which represents the pool of Nagios servers for Basic or Platinum MaaS subscriptions, respectively. The appropriated queue is selected according to the MaaS slice subscription stored in cloud slice attribute `@slice.MaaS`.

Configurator Workers are responsible for retrieving configurator calls from appropriated Configurator Queues (Figures 1 and 2, step 7). The Configurator Worker executes the configurator (step 8) and stores its execution status and output to send to FlexACMS Core through the REST Web service (step 9). This information is used by cloud administrators for future analysis or debugging. Configurator Workers are aware

TABLE I
EXAMPLE OF CONFIGURATORS ATTRIBUTES

Configurator	Attr.	Value
Name: nagios_host_basic Queue: nagios_basic	Int. Cond.	New Slice @slice.MaaS =~ /basic/
Name: nagios_cpu_basic Queue: nagios_basic	Int. Cond. Cond.	New Resource @resource.identifier =~ /CPU/ @slice.MaaS =~ /basic/
Name: nagios_host_plat Queue: nagios_platinum	Int. Cond.	New Slice @slice.MaaS =~ /platinum/
Name: nagios_cpu_plat Queue: nagios_platinum	Int. Cond. Cond.	New Resource @resource.identifier =~ /CPU/ @slice.MaaS =~ /platinum/

of the monitoring server load. Thus, they can decide stop retrieving configurator calls while the server load is unacceptable (Figure 2, phase 10). When the server load remains to an acceptable value, the Configurator Worker decides to consume configurator calls again (phase 11).

Configurator Queues and Configurator Workers enabled the two major features of enhanced FlexACMS. Firstly, the dynamic and automatic attribution of configuration tasks is achieved when the configurator calls are attributed to pools of monitoring servers (Configurator Queues) that are selected accordingly to predefined rules. In the previous version, the configurator tasks were statically attributed to a monitoring server. Secondly, load balancing is achieved when Configurator Workers concurrently consume the same Configurator Queue. Figure 2 presents the concurrency for configurator calls of two Nagios servers (Nagios#A and Nagios#B). This approach also enables that servers stop receiving monitoring tasks based on its load (Figure 2, phase 10) because Configurator Workers are aware of the server load.

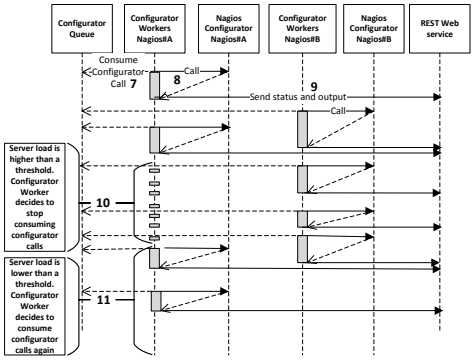


Fig. 2. Sequence Diagram for enhanced FlexACMS

B. Implementation details

We developed a gatherer to collect information about cloud slices for OpenStack [5] platform. The OpenStack gatherer uses several web services from OpenStack API [18] to retrieve information about cloud slices hosted in the platform. The response is organized according to the REST Web service definition. The OpenStack gatherer was developed in Python.

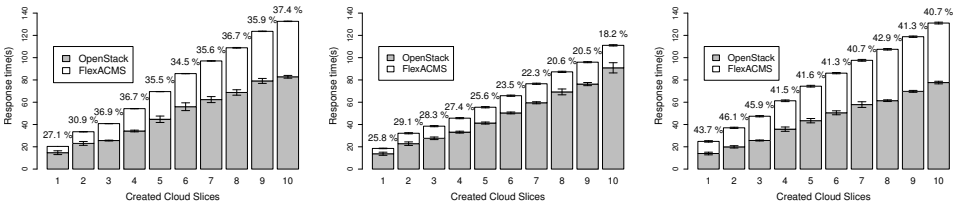
The FlexACMS Core was developed using the Ruby on Rails (RoR) framework which facilitates the development of REST Web services. RoR also facilitated the development of web-based interface used by administrators to interact with the FlexACMS. MySQL 5.5 is used as database to store cloud platform information. As the monitoring is performed by monitoring solutions and it is not performed by FlexACMS, we believe that a relational database is appropriated to this context. The queues are based on a RoR library called Resque that uses Redis, a key/value database, as backend to store the information related to queues and workers. Resque also allows the execution of workers based on Ruby code, which we used to develop Request Workers and Change Workers. Resque allows administrators to adjust the number of workers that will consume each queue. Based on this, the administrators can adjust the number of workers according to the capacity of the server hosting the FlexACMS framework (i.e, the framework server) and their response time requirements.

We developed the Configurator Worker using Perl, which consumes configurator calls within the queues directly from Redis. To be aware of the server load, the Configurator Worker reads the file `/proc/loadavg` that stores the average load of the server. We also developed a bash script to automate the initialization of configurator workers. This bash script accepts as arguments the number of workers that must be initialized, the queue name that they will consume, and the maximum load threshold that is acceptable in that server. By this manner, the administrators can also adjust the number of workers and the server load acceptable according to the monitoring server capacity and response time requirements.

Finally, we developed configurators for both Nagios and MRTG using Perl scripts. Nagios uses distinct configurations for host and service/resource status. We then define a configurator to reflect the creation of new slices (hosts) in Nagios, and a configurator for each monitored resource. We also developed a configurator for executing scripts that configure MRTG: `cfgmaker` and `indexmaker`, used to create configuration and index page required by MRTG graphs, respectively.

V. EVALUATION

The FlexACMS evaluation discussed in this section is twofold. First, we discuss the functional issues of employing FlexACMS in an IaaS environment with real cloud platforms and monitoring solutions. The goal is to identify the overhead introduced by FlexACMS when automatically configuring the monitoring slices required for cloud slices of a new IaaS service request from a cloud user. Second, we evaluate the scalability issues of the FlexACMS in order to determine the feasibility of our solution in large cloud environments. The numbers depicted in the graphics are the mean values observed in the experiments. All the experiments were repeated 30 times with confidence intervals of 95%. The details of these evaluations are described in the following subsections.



(a) Previous FlexACMS version (2 metrics) (b) Enhanced FlexACMS version (2 metrics) (c) Enhanced FlexACMS version (52 metrics)

Fig. 3. Comparative: serial vs. parallel approach

A. *Functional Evaluation*

We use the functional evaluation to analyze two issues. First we identify the overhead introduced by FlexACMS in automatically configure monitoring slices for cloud slices. The goal is to quantify the increase on the response time by adding the automatic configuration of the monitoring support for a cloud slice. Second, we compare the gains that enhanced architecture depicted on Figure 1 has over the previous architecture [3].

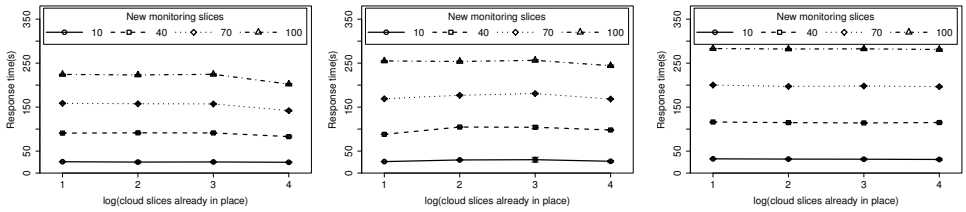
The evaluation scenario is composed of two servers: an OpenStack, and a monitoring server. The former has two Intel(R) Xeon(R) CPU E5430@2.66GHz with 4 cores per processor, 16GB of RAM, running Ubuntu Server 12.04 LTS. The latter has one Intel(R) Core(TM) i5 CPU 650@3.20GHz, 4GB of RAM, running Ubuntu Server 12.10. Both are connected to a switch by 1 Gbps links. The OpenStack server hosts all components of OpenStack infrastructure and an OpenStack gatherer. The monitoring server hosts the FlexACMS Core, Nagios and MRTG solutions, their corresponding configurators, and runs 10 configurator workers simultaneously.

We observe the time required to create a cloud slice using the OpenStack platform (OpenStack time) and the corresponding time to inform FlexACMS and to configure the monitoring slices associated to new cloud slices (FlexACMS time). The monitoring slices in the experiment reflects two metrics: host status and CPU usage. Thus, two configurators were defined: one for the host monitoring configuration in Nagios, and the second to create configuration to monitor CPU usage.

The Figure 3a shows the evaluation results of response time of the previous architecture. OpenStack requires around to 73-63% of the experiment time to create cloud slices. The previous version of FlexACMS required around to 37-27% of experiment time to configure the corresponding monitoring slices. The Figure 3b shows the response time results of the enhanced architecture. In this experiment, OpenStack requires around to 82-71% of the experiment time to create cloud slices, while the enhanced FlexACMS requires around to 29-18% of the experiment time to configure the corresponding monitoring slices. The enhanced FlexACMS reduces in circa of 10% the impact of the automatic monitoring slice configuration in the whole experiment time: OpenStack plus FlexACMS times.

In Figure 3b there is a clear trend showing that the increase in the number of cloud slices does not create a significant increase in the response time of the enhanced FlexACMS, when configuring few metrics per monitoring slice. For instance, when we analyze the bars for the creation of 10 monitoring slices, we observed that OpenStack requires around 81.8% of the experiment time, and FlexACMS requires 18.2% of the experiment time. In addition, when we compare the experiment time for creation of 9 monitoring slices (*i.e.*, cloud slices), we observe again that the response time of enhanced FlexACMS corresponds to 20.5%, almost the same value if compared with the case of 10 monitoring slices. This happens because the enhanced FlexACMS is built to better distribute the configuration load by exploring the parallelism and thus reducing the overall time of configuration of multiple monitoring slices. These experiments show that enhanced FlexACMS architecture fits well when a highly number of monitoring slices need to be created with few number of monitoring metrics per slice.

However, the monitoring slices were composed of only two metrics: host status and CPU usage. We then observed FlexACMS configuring more complex monitoring slices, which are composed of: a host status and 50 metrics monitored by Nagios, and a network usage graphic built by MRTG. In this case, each monitoring slice has 52 metrics and requires 52 configurator calls to be configured. The Figure 3c shows that FlexACMS requires more time to configure more complex monitoring slices. In this scenario, OpenStack requires around to 60-55% of the experiment time to create cloud slices against 45-40% required by FlexACMS to configure the corresponding monitoring slices. Indeed, looking to the bars related with the 10 created cloud slices in Figures 3b and 3c, the FlexACMS time to configure more complex monitoring slices was 2.66 times (50.4s in Figure 3c) higher than the response time to configure simpler monitoring slices (20.1s in Figure 3b). Thus, we can conclude that the number of metrics of a monitoring slice affects the FlexACMS time. Although, the response time and the monitoring slice size did not increase in the same rate (respectively, 2.66 times vs. 26 times). The scalability evaluation on Section V-B will further analyze a growing number of metrics per monitoring slice.



(a) 5 metrics per monitoring slice (b) 25 metrics per monitoring slice (c) 50 metrics per monitoring slice

Fig. 4. Scalability results

B. Scalability Evaluation

In this scalability evaluation, we are interested in further analyzing how the response time is affected when a higher number of metrics is used in a single monitoring slice. The evaluation scenario was composed of two servers: a monitoring server, and a FlexACMS server. Both servers have one Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz with 4GB of RAM, running Gentoo Linux and Ubuntu Server 12.10, respectively. Both servers are connected to a switch with links of 1 Gbps. The monitoring server runs 10 configurator workers simultaneously, and hosts Nagios configurators. The FlexACMS server hosts the FlexACMS Core, the gatherer that generates artificial input, and runs 10 workers simultaneously that are able to consume Requests queue and Changes queue. The implementation details, except the gatherer, are described in Section IV-B. The gatherer was developed using Perl to generate information about cloud slices with 5 resources and 20 associated information. These values are similar to the values achieved when we used the real OpenStack gatherer.

We varied the number of metrics in each monitoring slice in 5, 25, and 50 metrics per monitoring slice, which represents 1, 5, and 10 metrics per resource, respectively. We believe that 50 metrics per monitoring slice (10 metrics per resource) is a high number of metrics. Another scalability issue that we are interested is the capacity of the enhanced FlexACMS to hold the response time when a large number of cloud slices are already in place. We varied the number of cloud slices in 10, 100, 1000, 10000 cloud slices already in place. As far as we know there are not statistics of real large IaaS scenarios. We believe that an InP that hosts 10000 cloud slices is large. We also varied the number of monitoring slices to be built during a burst in 10, 40, 70, and 100 new monitoring slices per burst. We believe that the creation of 100 new cloud slices in a single gatherer request is a high number of new cloud slices. Specially, if we consider that gatherer requests to FlexACMS will be into a polling process of few minutes.

The first observation when analyzing the response time evaluation results shown in Figures 4a, 4b, and 4c is related to the number of cloud slices that are already in place. We can observe this following the lines along the x-axis in each figure. The x-axis uses a logarithmic scale to allow a proper comparison. We vary the number of cloud slices from 10 to 10000 cloud slices, *i.e.*, from 10^1 to 10^4, and we can observe that lines are flat, and are almost constant along the x-axis. This means that FlexACMS scales when a growing number of cloud slices are in place. The second observation is related to the number of monitoring slices that need to be built in a burst. We can observe this comparing the distance between adjacent lines into the same figure. We varied the number in 10, 40, 70, and 100 required monitoring slices. Thus, we varied by a fixed number (30 monitoring slices), and thus the response time also must vary by a fixed value between lines in the same figure. We can observe that the lines keep a similar distance when increase the number of monitoring slices.

However, it is difficult to measure that distances are similar just looking to the figures. Therefore, we calculated the throughput for each scenario to assure that FlexACMS scales when the number of monitoring slices increases. To calculate the throughput we need the number of configured metrics for each line, which can be calculated multiplying the number of new slices by the number of metrics per slice. For instance, 100 new slices $*$ 50 metrics per slice $=$ 5000 metrics to be configured. We just need to divide the response time by the number of metrics configured to achieve the throughput. The lines on each figure has similar throughput that are around to 2.5, 10, and 17 metrics configured per second in Figures 4a, 4b, and 4c, respectively. We observed that in the same figure the observations have similar throughput, thus, we can conclude that FlexACMS scales when the number of monitoring slices that need to be built increases.

Finally, we observe the effect of growing number of metrics in each monitoring slice by comparing the lines of same pattern between neighbor figures. We varied the number of metrics per slice in 5 metrics, 25 metrics, and 50 metrics. We can observe, for instance, comparing the lines of Figures 4a and 4b which present results for 5 metrics and 25 metrics, respectively, *i.e.* monitoring slices 5 times greater. The response time in Figure 4b is slight higher than in Figure 4a, and the difference is much smaller than 5 times. Similar conclusions can be done analyzing the difference between

Figures 4a and 4c, which present results for 5 and 50 metrics, respectively, *i.e.* monitoring slices are 10 times greater, and between Figures 4b and 4c, which present results for 25 and 50 metrics, respectively, *i.e.*, where monitoring slices are 2 times greater. We observed, thus, that FlexACMS scales when a growing number of metrics are used on monitoring slices.

The enhanced FlexACMS architecture can consume more computational resources that our previous single-threaded architecture. However, the resource consumption can be tuned by the administrator adjusting the number of workers. Our experiments were performed in commodity computers with 2 cores and 2 threads per core; however, we conclude that achieved response times are appropriate. Beyond, despite our evaluated scenarios were based on single machine as FlexACMS server, its architecture uses components employed in traditional environments, such as web server, database, and queue system. Thus, there are solutions to leverage the throughput of these components, such as load balancers. Cloud administrators can built more complex monitoring environments employing load balancers to increase FlexACMS throughput if necessary.

VI. Conclusion

In this paper, we enhanced FlexACMS which configures monitoring slices automatically when cloud slices are created. The enhancement introduced two features that leverages the automation achieved by FlexACMS. The first feature is the dynamic and automatic attribution of configuration tasks to pools of monitoring servers. The second feature is the load balancing when attributing configuration tasks among the monitoring servers members of a pool. This feature is achieved because monitoring servers volunteer to receive configuration tasks and can decide to stop receiving tasks while they are overloaded. Both features required architectural modifications into the FlexACMS Core, such as the use of queue/workers. This new approach allowed FlexACMS to break the whole task of configuring monitoring slices into small tasks performed in parallel improving its response time.

We conducted a comparative evaluation between previous and enhanced FlexACMS. We measured the experiment time as the time to create cloud slices in OpenStack and the additional time required to configure the corresponding monitoring slices in FlexACMS. The previous version configured monitoring slices in 34,7% of the experiment time. The enhanced version configured the monitoring slices in 24,1% of the experiment time. Thus, the enhanced version reduces in circa of 10% its influence in the experiment time. However, when looking to FlexACMS time to configure 10 monitoring slices, the enhanced version reduces the response time from 49,76s to 20.1s, which represents up to 60% of response time reduction. In this case, FlexACMS explores the parallelism available when needs to configure 10 monitoring slices, which demonstrate the benefit of queue/workers approach.

We also conducted a scalability evaluation, which focuses on how the FlexACMS response time is affected by different aspects. We varied the number of cloud slices already in place, the number of monitoring slices to be built, and the number of metrics in each monitoring slice. We concluded that the response time was not affected by the number of cloud slices already in place. We concluded that the response time is affected by the number of metrics in each monitoring slice and by the number of monitoring slices that must be built, but the response time does not increase in the same rate that the number of metrics and monitoring slices increases. These observations show that the enhanced FlexACMS scales in regards to response time, which demonstrate the feasibility of our approach in large cloud computing environments.

After solve the problem of create monitoring slices automatically, we are able to address other related cloud monitoring issues, as the reconfiguration and destruction of monitoring slices, and adaptive allocation strategies in placing new monitoring slices. We also plan to employ FlexACMS into an InP provider to improve its capabilities and offer it as an open source project. Furthermore, we want to evaluate the feasibility of the framework ideas in Platform as a Service (PaaS) and Software as a Service (SaaS) cloud models.

References

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A Break in the Clouds: Towards a Cloud Definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, Dec. 2008.

[2] S. Meng and L. Liu, "Enhanced monitoring-as-a-service for effective cloud management," *Computers, IEEE Transactions on*, vol. 62, no. 9, pp. 1705–1720, 2013.

[3] M. Carvalho, R. Esteves, G. Rodrigues, L. Z. Granville, and L. M. R. Tarouco, "A Cloud Monitoring Framework for Self-Configured Monitoring Slices Based on Multiple Tools," in *9th Intl. Conference on Network and Service Management 2013 (CNSM 2013) - Poster Session*, Oct 2013.

[4] J. Montes, A. Sánchez, B. Memishi, M. S. Pérez, and G. Antoniu, "GMonE: A complete approach to cloud monitoring ," *Future Generation Computer Systems*, pp. –, 2013.

[5] OpenStack Community, "OpenStack," 2014, available at: http://www.openstack.org/ accessed in: Feb. 2014.

[6] Nagios Enterprises, "Nagios," 2014, available at: http://www.nagios.org/ accessed in: Feb. 2014.

[7] Tobias Oetiker, "MRTG," 2014, available at: http://oss.oetiker.ch/mrtg/ accessed in: Feb. 2014.

[8] S. De Chaves, R. Uriarte, and C. Westphall, "Toward an Architecture for Monitoring Private Clouds," *Communications Magazine, IEEE*, vol. 49, no. 12, pp. 130 –137, Dec. 2011.

[9] Amazon, "Amazon CloudWatch," 2014, available at: http://aws.amazon.com/en/cloudwatch/ accessed in: Feb. 2014.

[10] OpenStack Community, "Ceilometer," 2014, available at: https://wiki.openstack.org/wiki/Ceilometer accessed in: Feb. 2014.

[11] Amazon, "Amazon Web Services," 2014, available at: http://aws.amazon.com/ accessed in: Feb. 2014.

[12] J. Shao, H. Wei, Q. Wang, and H. Mei, "A Runtime Model Based Monitoring Approach for Cloud," in *IEEE 3rd International Conference on Cloud Computing*, Jul. 2010, pp. 313 –320.

[13] M. Rak, S. Venticinque, T. Mahr, G. Echevarria, and G. Esnal, "Cloud Application Monitoring: The mOSAIC Approach," in *IEEE CloudCom 2011*, Dec. 2011, pp. 758 –763.

[14] H. Han, S. Kim, H. Jung, H. Yeom, C. Yoon, J. Park, and Y. Lee, "A RESTful Approach to the Management of Cloud Infrastructure," in *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, Sep. 2009, pp. 139 –142.

[15] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093 – 2115, 2013.

[16] Puppet Labs, "Puppet," 2014, available at: http://puppetlabs.com/ accessed in: Feb. 2014.

[17] Opscode, "Chef," 2014, available at: http://www.opscode.com/chef/ accessed in: Feb. 2014.

[18] OpenStack Community, "OpenStack API," 2014, available at: http://api.openstack.org/ accessed in: Feb. 2014.