

Marcio Veronez

Um Catálogo de Requisitos Não-Funcionais e Padrões Arquiteturais para Microsserviços

Cascavel-PR

2021

Lista de ilustrações

Figura 1 – SIG para o RNF Agility	57
Figura 3 – SIG para o RNF Deployability	58
Figura 4 – SIG para o RNF Independency	59
Figura 5 – SIG para o RNF Maintainability	60
Figura 6 – SIG para o RNF Manageability	61
Figura 7 – SIG para o RNF Monitorability	62
Figura 8 – SIG para o RNF Performance	63
Figura 9 – SIG para o RNF Portability	64
Figura 10 – SIG para o RNF Reusability	65
Figura 11 – SIG para o RNF Scalability	66
Figura 12 – SIG para o RNF Security	67
Figura 2 – SIG para o RNF Availability	68

Listas de tabelas

Tabela 1 – RNF Agilidade e seus descendentes	7
Tabela 2 – RNF Disponibilidade e seus descendentes	8
Tabela 3 – RNF Facilidade de implantação e seus descendentes	9
Tabela 4 – RNF Evolucionalidade e seus descendentes	9
Tabela 5 – RNF Independência e seus descendentes	10
Tabela 6 – RNF Manutenibilidade e seus descendentes	10
Tabela 7 – RNF Gerenciabilidade e seus descendentes	11
Tabela 8 – RNF Monitorabilidade e seus descendentes	11
Tabela 9 – RNF Performance e seus descendentes	12
Tabela 10 – RNF Portabilidade e seus descendentes	13
Tabela 11 – RNF Reusabilidade e seus descendentes	13
Tabela 12 – RNF Escalabilidade e seus descendentes	14
Tabela 13 – RNF Segurança e seus descendentes	14
Tabela 14 – Padrões arquiteturais <i>Adapters</i> e RNFs impactados	17
Tabela 15 – Padrões arquiteturais para APIs e RNFs impactados	21
Tabela 16 – Padrões arquiteturais para automação e RNFs impactados	23
Tabela 17 – Padrões arquiteturais para balanceamento de carga e RNFs impactados	24
Tabela 18 – Padrões arquiteturais para bancos de dados e RNFs impactados	28
Tabela 19 – Padrões arquiteturais para cache de resultados e RNFs impactados . .	30
Tabela 20 – Padrões arquiteturais para configuração e RNFs impactados	31
Tabela 21 – Padrões arquiteturais para decomposição dos serviços e RNFs impactados	33
Tabela 22 – Padrões arquiteturais para descoberta de serviços e RNFs impactados .	34
Tabela 23 – Padrões arquiteturais para escala dos serviços e RNFs impactados . .	36
Tabela 24 – Padrões arquiteturais para implantação dos serviços e RNFs impactados	38
Tabela 25 – Padrões arquiteturais para integração entre serviços e RNFs impactados	43
Tabela 26 – Padrões arquiteturais para migração e RNFs impactados	44
Tabela 27 – Padrões arquiteturais para monitoramento e controle de falhas e RNFs impactados	50
Tabela 28 – Padrões arquiteturais para registro de serviços e RNFs impactados . .	52
Tabela 29 – Padrões arquiteturais para reúso e RNFs impactados	53
Tabela 30 – Padrões arquiteturais para segurança e RNFs impactados	55

Sumário

1	O CATÁLOGO	7
1.1	RNFs importantes no processo de migração de sistemas monolíticos para MSs	7
1.2	Padrões Arquiteturais e seus Impactos sobre os RNFs	14
1.2.1	Padrões <i>Adapters</i>	15
1.2.2	Padrões para APIs	17
1.2.3	Padrões para Automação	22
1.2.4	Padrões para Balanceamento de Carga	24
1.2.5	Padrões para Bancos de Dados	25
1.2.6	Padrões para Cache de Resultados	29
1.2.7	Padrões para Configuração	30
1.2.8	Padrões para Decomposição dos Serviços	31
1.2.9	Padrões para Descoberta de Serviços	34
1.2.10	Padrões para Escala dos Serviços	35
1.2.11	Padrões para Implantação dos Serviços	36
1.2.12	Padrões para Integração entre Serviços	38
1.2.13	Padrões para Migração	43
1.2.14	Padrões para Monitoramento e controle de falhas	44
1.2.15	Padrões para Registro de Serviços	51
1.2.16	Padrões para Reúso de MS	53
1.2.17	Padrões para Segurança	53
1.3	Grafos de Interdependência de softgoals - SIGs	55
	REFERÊNCIAS	69
	FONTES CONSIDERADAS NA IDENTIFICAÇÃO DOS PADRÕES ARQUITETURAIS	71
	FONTES CONSIDERADAS NA COMPLEMENTAÇÃO DAS INFORMAÇÕES SOBRE OS PADRÕES ARQUITETURAIS	73

1 O catálogo

O catálogo é composto por:

(i) uma listagem de RNFs importantes no domínio de migração de sistemas monolíticos para MSs e seus RNFs descendentes; (ii) uma listagem de padrões arquiteturais e respectivas contribuições positivas e negativas sobre os RNFs descendentes; (iii) grafos de interdependência de *softgoals* (SIGs), mostrando, para cada RNF principal, seus RNFs descendentes e possíveis operacionalizações através dos padrões arquiteturais.

A seguir são apresentados cada um destes itens.

1.1 RNFs importantes no processo de migração de sistemas monolíticos para MSs

A seguir são apresentados os RNFs importantes nos processos de migração de sistemas monolíticos para MSs. Para cada um destes requisitos, são apresentados seus RNFs descendentes.

Agilidade de uma arquitetura de software pode ser descrita como um sistema que é construído utilizando uma coleção de componentes pequenos e isolados, os quais podem ser desenvolvidos por times específicos com experiência e conhecimentos necessários para desenvolvê-los da melhor forma. Esta independência possibilita uma liberdade de escolha de tecnologias, favorece o desenvolvimento incremental e permite que módulos sejam testados e substituídos isoladamente. A arquitetura de MSs, devido às suas características, é uma arquitetura que possibilita o incremento da agilidade no desenvolvimento de software [Mišić, Ramač e Mandić 2017].

Tabela 1 – RNF Agilidade e seus descendentes

Agility	
RNF descendente	Descrição
Less development complexity	Diminuição da complexidade de desenvolvimento
Easier database management	Facilidade de gerenciamento das bases de dados
Better service integration	Melhor integração entre serviços
Use of service templates	Rapidez na criação de novos serviços através da utilização de templates de serviços
Automated deploy	Automatização das atividades de implantação
Automated test	Automatização de testes
Service orchestration	Orquestração de serviços através da utilização de ferramentas de gerenciamento
Continua na próxima página	

Tabela 1 – continuado da página anterior

RNF descendente	Descrição
Time to market reduction	Diminuição no tempo de lançamento ou atualização de serviços
Cost reduction	Diminuição no custo de desenvolvimento e manutenção de serviços

Fonte: Produzido pelo autor

Disponibilidade é a capacidade que um sistema tem de se recuperar de falhas, de forma que o tempo de indisponibilidade de uma função ou serviço não exceda um valor predefinido [Bass, Clements e Kazman 2003]. Normalmente, provedores de serviços definem em contrato a garantia de disponibilidade de seus serviços [O'Brien, Merson e Bass 2007]. A disponibilidade é um atributo importante, pois MSSs são distribuídos por natureza e vulneráveis a várias falhas, cujas causas são difíceis de serem analisadas devido às relações entre os MSSs [Li et al. 2021].

Tabela 2 – RNF Disponibilidade e seus descendentes

Availability	
RNF descendente	Descrição
Load balancing	Utilização de servidores que fazem o redirecionamento das requisições aos serviços de forma balanceada, evitando a sobrecarga e indisponibilidade dos serviços
Support older API versions	APIs legadas devem ser suportadas até que seu uso seja descontinuado por todos os clientes
Containerized services	A execução de serviços em containers permite utilizar melhor a infraestrutura, facilitando a escalabilidade e isolamento, levando a uma maior disponibilidade dos serviços
State availability	As interações anteriores e atuais dos usuários devem ser persistidas e estarem sempre disponíveis aos serviços
Fail isolation	Falhas devem ser isoladas para que não sejam propagadas ao sistema, prejudicando seu funcionamento
Message broking	Canais de comunicação entre clientes e serviços são definidos e requisições são enfileiradas e distribuídas de acordo com políticas estabelecidas, objetivando obter desempenho e disponibilidade dos serviços
Descentralized endpoint	Pontos de acesso a serviços devem ser descentralizados, para que a falha de um ponto não cause impactos no sistema
Service registry	Serviços devem ser registrados em um servidor de registros, o qual será consultado pelos clientes para obter os endereços atuais dos serviços disponíveis
Database availability	Sistemas de bancos de dados devem estar sempre disponíveis, tratando possíveis inconsistências causadas pela descentralização ou interrupção momentânea do serviço de banco de dados
Reduce service downtime when deploying	O deploy de novas versões de serviços não deve interromper sua utilização por parte dos clientes

Fonte: Produzido pelo autor

Facilidade de Implantação é uma característica que indica o quanto facilitada é a disponibilização de uma nova versão de um sistema ou serviço para os usuários [Bass, Clements e Kazman 2003]. O objetivo principal de uma implantação é minimizar os possíveis impactos para os usuários, evitando falhas ou quedas no serviço [Bass, Weber e Liming 2015].

Tabela 3 – RNF Facilidade de implantação e seus descendentes

Deployability	
RNF descendente	Descrição
Decomposed services	Serviços devem ter uma correta granularidade para que sejam implantados de forma eficiente e independente
Efficient deploy strategy	Serviços devem ser executados em estruturas que facilitem a implantação dos serviços
Automated deploy	Utilização de ferramentas para implantação automatizada de serviços
Orchestrated deploy	Utilização de ferramentas para gerenciamento e orquestração das implantações

Fonte: Produzido pelo autor

Evolucionalidade diz respeito ao grau de efetividade e eficiência com que um software pode ser adaptado ou estendido, sendo bastante importante em sistemas com mudanças frequentes, como os baseados em internet [Bogner et al. 2019].

Tabela 4 – RNF Evolucionabilidade e seus descendentes

Evolvability	
RNF descendente	Descrição
Legacy software integration	Sistemas em MSs devem permitir a integração com sistemas legados
Service extension	Serviços devem ser extensíveis
Easier service replacement	Serviços devem ser facilmente substituíveis
API documentation	A documentação da API fornecida pelos serviços deve estar disponível aos clientes e desenvolvedores
API evolution	A API fornecida pelos serviços deve ter sua evolução facilitada
API versioning	A API fornecida pelos serviços deve ter informação sobre versionamento, para que desenvolvedores possam lançar novas versões, mantendo em execução versões ainda utilizadas
Database evolution	Os bancos de dados utilizados pelos serviços devem poder evoluir independente de outros serviços
Legacy software migration	Software legado deve ser migrado para a arquitetura de MSs

Fonte: Produzido pelo autor

Independência, no contexto de MSs, diz respeito à modularidade, alta coesão e baixo acoplamento entre serviços, permitindo que sejam implantados, escalados e desenvolvidos de forma isolada, permitindo a utilização de tecnologias mais adequadas para cada MS [Taibi e Lenarduzzi 2018]. Os serviços que compõe um sistema devem ser responsáveis por tarefas específicas [Vural e Koyuncu 2021].

Tabela 5 – RNF Independência e seus descendentes

Independency	
RNF descendente	Descrição
Separated services code	Serviços não devem ter compartilhamento de código fonte
Isolated database	O banco de dados utilizado por um serviço não deve ser compartilhado
Isolated service instance	A instância de um serviço deve ser executada de forma isolada
Decomposed services	A decomposição correta dos serviços diminui sua dependência com relação a outros componentes e serviços

Fonte: Produzido pelo autor

Manutenibilidade pode ser descrita com o grau de efetividade e eficiência em que um sistema pode ser modificado [Bass, Clements e Kazman 2003]. Tais modificações incluem correção de defeitos ou implementação de melhorias no software [Mairiza, Zowghi e Nurmuliani 2010].

Tabela 6 – RNF Manutenibilidade e seus descendentes

Maintainability	
RNF descendente	Descrição
Documented API	APIs bem documentadas facilitam tarefas de manutenção
Decoupled services	Serviços desacoplados melhoram a manutenibilidade, pois tem seu código fonte independente de outros serviços
Automated test	A automatização de testes facilita e agiliza a manutenção dos sistemas
Database segregation	A separação das bases de dados utilizadas pelos serviços permitem a realização de alterações e correções sem impactar outros serviços
Service templates	A utilização de templates para criação dos serviços traz uma melhor padronização e facilita futuras manutenções
Decomposed services	Serviços corretamente compostos permitem que sejam feitas alterações e correções em um serviço sem afetar os demais

Fonte: Produzido pelo autor

Gerenciabilidade diz respeito à capacidade dos MSs em serem gerenciáveis e é indicada pelo grau de centralização do gerenciamento, onde um menor grau é o ideal (auto-gerenciamento) [Cojocaru, Oprescu e Uta 2019]. Aplicações em nuvem, como MSs,

requerem uma gerência contínua para provisionar recursos de forma que os serviços continuem atendendo os usuários de forma adequada, mitigando possíveis falhas [Toffetti et al. 2015].

Tabela 7 – RNF Gerenciabilidade e seus descendentes

Manageability	
RNF descendente	Descrição
Ease of configuration	A facilidade de configuração dos serviços facilita seu gerenciamento
Service orchestration	A utilização de ferramentas para organização dos serviços facilita o gerenciamento do sistema
Automated deploy	A utilização de ferramentas para implantação automatizada auxilia na melhoria do gerenciamento das implantações
Automated test	A utilização de ferramentas para execução de testes automatizados auxilia no gerenciamento e eficiência dos testes
Easier database management	Os bancos de dados utilizados pelos serviços devem ter seu gerenciamento facilitado
Easier security management	O gerenciamento da segurança dos serviços deve ser facilitado

Fonte: Produzido pelo autor

Monitorabilidade é a capacidade que um sistema tem de permitir que seja monitorado enquanto esteja em execução [Bass, Clements e Kazman 2003]. No caso de MSs, alguns itens que podem ser monitorados são referentes à infraestrutura (p. ex., *containers*), aplicação (p. ex., tempo de resposta) e informações sobre o ambiente (p. ex., rede de dados). A monitorabilidade de MSs é uma parte essencial e complexa, devido as suas estruturas dinâmicas e comportamentos variados [Li et al. 2021].

Tabela 8 – RNF Monitorabilidade e seus descendentes

Monitorability	
RNF descendente	Descrição
Communication monitoring	A comunicação interna (entre componentes de um serviço) e externa (entre diferentes serviços) deve ser monitorada
Resources use monitoring	A utilização dos recursos de hardware deve ser monitorada
Monitored data processing	Os dados coletados durante o monitoramento devem ser processados para que sua análise seja facilitada
Monitored data storing	Os dados coletados durante o monitoramento devem ser armazenados para posterior análise
Security monitoring	Os serviços devem ser monitorados quanto a sua segurança, com o objetivo de mitigar ameaças
Traceability	A requisição realizada por um cliente deve ser rastreada entre um ou mais serviços em que é executada

Fonte: Produzido pelo autor

Performance é a habilidade que um sistema tem em atender requisitos de tempo para responder a um evento [Bass, Clements e Kazman 2003]. Exemplos de performance são o tempo que um sistema leva para processar uma requisição, quantas requisições podem ser processadas em um intervalo de tempo ou a capacidade de atender uma requisição em um tempo máximo estabelecido [O'Brien, Merson e Bass 2007]. Performance é um atributo significativo devido à característica distribuída dos MSs [Li et al. 2021].

Tabela 9 – RNF Performance e seus descendentes

Performance	
RNF descendente	Descrição
Load balancing	A distribuição balanceada das requisições entre as instâncias de um serviço permite uma utilização eficiente dos recursos e diminuição do tempo de resposta
Efficient message processing	O processamento em paralelo permite uma utilização melhor dos recursos e diminuição no tempo de resposta
Efficient resources utilization	A utilização eficiente dos recursos melhora a rapidez com que as tarefas são executadas pelos serviços
Efficient message exchange	A troca de mensagens entre os serviços deve ser feita de forma a minimizar a quantidade de informações trafegadas e o tempo necessário para processamento e envio.
Efficient database query	O processamento de consultas às bases de dados utilizadas pelos serviços deve ser eficiente
Efficient host communication	A comunicação entre um serviço e seu host hospedeiro deve ser eficiente
Efficient shared components communication	Um serviço deve se comunicar de forma eficiente com outros componentes que são compartilhados entre vários serviços
Metrics measurement	Devem ser utilizadas métricas para avaliação dos aspectos de performance dos serviços
Efficient health check	Serviços devem ser periodicamente checados para verificação de sua disponibilidade

Fonte: Produzido pelo autor

Portabilidade se refere à facilidade com que um software que foi desenvolvido para ser executado em uma plataforma possa ser alterado para ser executado em uma plataforma diferente [Bass, Clements e Kazman 2003]. Os MSs auxiliam na portabilidade de software, pois grande parte do sistema é encapsulado no serviço (que expõe sua API a outros serviços que foram desenvolvidos em diversas plataformas), está distribuído e executado na nuvem, restando geralmente apenas a interface de usuário, que poderá ser portada para múltiplos dispositivos de forma mais simples.

Tabela 10 – RNF Portabilidade e seus descendentes

Portability	
RNF descendente	Descrição
Services portability	Serviços devem ser executados em múltiplos ambientes sem necessidade de modificações ou recompilações
Services compatibility	Serviços devem ser compatíveis com necessidades específicas dos clientes, que poderá necessitar de APIs customizadas para utilização em diferentes dispositivos
Legacy code compatibility	Sistemas legados devem ser migrados para MSs gradualmente, mantendo a compatibilidade com o código antigo até que todo o sistema tenha sido reimplementado

Fonte: Produzido pelo autor

Reusabilidade de software pode ser vista como o uso de componentes de software existentes para a construção de novos sistemas, compreendendo código-fonte, requisitos, especificações e qualquer informação necessária para a construção de um software [Ruben Prieto-Díaz 1993]. Sendo assim, a reusabilidade é a capacidade que um sistema, módulo ou serviço possui de ser reutilizável sem que seja necessário grande esforço por parte dos desenvolvedores [Mendonça et al. 2018]. Os MSs, devido à sua característica de independência e coesão, podem ser reutilizados por diversas aplicações [Da Silva et al. 2019].

Tabela 11 – RNF Reusabilidade e seus descendentes

Reusability	
RNF descendente	Descrição
Service templates	Reúso e padronização na criação de novos serviços através da utilização de templates de serviços

Fonte: Produzido pelo autor

Escalabilidade é a capacidade que um sistema tem de funcionar corretamente quando a quantidade de recursos disponíveis para o sistema é alterada com o objetivo de atender às expectativas dos usuários [O'Brien, Merson e Bass 2007]. A escalabilidade pode ser *horizontal*, quando são atribuídos mais recursos à unidades lógicas, como, por exemplo, adicionar mais servidores a um *cluster*) ou então *vertical*, quando são atribuídos mais recursos à unidades físicas, como, por exemplo, adicionar mais memória a um único computador [Bass, Clements e Kazman 2003]. A escalabilidade é um atributo importante para MSs, pois a própria arquitetura tem como objetivo permitir que sistemas sejam executados em nuvem de forma escalável.

Tabela 12 – RNF Escalabilidade e seus descendentes

Scalability	
RNF descendente	Descrição
Database scalability	Os bancos de dados utilizados pelos serviços devem ser escaláveis
State scalability	As interações anteriores e atuais dos usuários devem ser persistidas em um meio de armazenamento escalável
Auto-scaling	A escala dos serviços deve ser automática
Decomposed services	A correta granularidade dos serviços permite que sejam melhor escalados
Lightweight runtime environment	O ambiente em que os serviços são executados deve ser leve, facilitando a escala dos mesmos

Fonte: Produzido pelo autor

Segurança é a habilidade que um sistema tem de proteger dados e informação, bloqueando o acesso não autorizado e permitindo o acesso de sistemas ou pessoas autorizadas [Bass, Clements e Kazman 2003]. É um atributo importante para MSs, pois, devido a sua característica distribuída, cada serviço pode estar vulnerável, se tornando uma possível porta de entrada para ataques [Li et al. 2021].

Tabela 13 – RNF Segurança e seus descendentes

Security	
RNF descendente	Descrição
Configuration security	A configuração dos serviços deve ter seu acesso restringido
Data security	O acesso aos dados gerenciados por um serviço deve ser controlado
Plataform security	A plataforma onde um serviço é executado deve ser independente, melhorando a segurança
Attack detection	Os serviços devem ser monitorados visando detectar possíveis ataques contra sua segurança
Communication security	A comunicação entre os serviços e seus clientes deve ser segura

Fonte: Produzido pelo autor

1.2 Padrões Arquiteturais e seus Impactos sobre os RNFs

A seguir são apresentados os padrões arquiteturais identificados. Para facilitar a compreensão, os padrões foram classificados pelo aspecto de que tratam no domínio dos MSs: *adaptadores, API, automação, balanceamento, bancos de dados, cache de resultados, configuração, decomposição, descoberta de serviços, escala, implantação, integração, migração, monitoramento, registro de serviços, reúso e segurança*. Ao final de cada categoria é apresentada uma tabela com a listagem de padrões arquiteturais, os RNFs impactados por

cada padrão e o nível de impacto. As informações contidas nestas tabelas são utilizadas para construção de SIGs, os quais são apresentados em seguida.

1.2.1 Padrões *Adapters*

Nesta categoria estão os padrões utilizados para implementação de algum tipo de adaptação na interface dos serviços disponibilizados.

Adapter

Contexto: MSs precisam incorporar serviços existentes (p. ex., SOAP, JMS ou serviços baseados em *mainframe*), mas as APIs destes serviços não são consistentes com a arquitetura de MSs.

Problema: Como realizar a tradução destes serviços existentes em APIs de MSs?

Solução: Construir adaptadores (*adapters*) que convertem as APIs existentes para APIs que possam ser utilizadas pelos clientes dos MSs. Este padrão é utilizado em casos em que os serviços existentes funcionam muito bem e não há necessidade de trocá-los, ou quando a mudança destes serviços pode prejudicar clientes existentes. Frequentemente é utilizado em conjunto com o padrão Results Cache, para reduzir o número de chamadas aos serviços legados.

Vantagens: Favorece a evolucionabilidade, permitindo que sistemas sejam modernizados para MSs sem que seja necessário reimplementar todo o código legado.

Aggregator

Contexto: Em um ambiente de MSs, vários serviços são responsáveis por tarefas específicas.

Problema: Como agregar o resultado de várias chamadas a diversos MSs, para servirem a uma funcionalidade de negócios maior ou mais complexa.

Solução: Implementar MSs agregadores, que são responsáveis por receber requisições, chamar outros MSs, combinar os resultados e então responder a requisição inicial.

Vantagens: Devido a simplicidade da API agregadora, facilita a implementação de novas funcionalidades que podem ser disponibilizadas em menor tempo.

Desvantagens: Pode aumentar a latência devido à implementação de uma nova camada intermediária.

Anti-corruption layer

Contexto: Em um ambiente de migração para MSs, podem existir funcionalidades legadas que não podem ser reimplementadas mas que ainda devem ser acessíveis pelos MSs, fazendo com que o novo sistema tenha que ser implementado utilizando as mesmas tecnologias legadas, dessa forma corrompendo a arquitetura de MSs.

Problema: Como fazer com que os MSs tenham acesso às funcionalidades legadas sem terem que utilizar as mesmas tecnologias do software legado?

Solução: Implementar uma camada anti-corrupção, com o objetivo de isolar a implementação legada dos MSs. Esta camada é responsável por receber as requisições dos MSs, fazer a tradução para o formato de mensagem esperado pelo sistema legado, posteriormente receber o retorno, traduzi-lo e envia-lo para os MSs.

Vantagens: Favorece a compatibilidade da nova arquitetura de MSs com sistemas legados. Permite isolar erros, auxiliando na resiliência dos MSs. *Desvantagens:* A implementação da camada anti-corrupção requer a inclusão de novos serviços, podendo aumentar a latência devido a implementação dessa nova camada intermediária.

API gateway

Contexto: MSs provêm suas funções para outros serviços através de APIs. A criação de aplicações baseadas na composição de diferentes MSs requer um mecanismo de agregação server-side.

Problema: Como implementar um mecanismo de agregação de MSs server-side?

Solução: Implementar uma API gateway, responsável por ser o ponto de entrada que direciona as requisições para os MSs corretos, invocando múltiplos serviços, agregando resultados, transformando protocolos, cuidando da autenticação e limitando tráfego entre clientes e serviços.

Vantagens: Facilita a extensão do sistema através da customização de APIs. Permite manter a compatibilidade com sistemas legados, dessa forma favorecendo a evolução dos novos sistemas.

Desvantagens: A utilização deste padrão pode prejudicar a *disponibilidade*, pois a utilização de um *gateway* central de acesso para todas as requisições representa um risco, caso esse ponto apresente falha. A *performance* dos serviços pode diminuir, devido a inclusão de novas operações de rede, como gerenciamento de segurança.

Backend for frontend

Contexto: Microsserviços que encapsulam funções no domínio de negócios não são claramente mapeados para necessidades específicas dos clientes. Por exemplo, os clientes podem ter diversos tipos de front-end, como mobile e web, cada um com necessidades específicas de acesso a funções e dados.

Problema: Como representar interfaces de serviço que sejam consistentes com a arquitetura de MSs mas que sejam adaptáveis às necessidades de cada tipo de cliente?

Solução: Construir diferentes interfaces personalizadas para cada tipo de cliente, agregando somente o que é necessário para atendê-lo. Um BFF não deve conter nenhuma

lógica de negócios, sendo geralmente desenvolvida pelo mesmo time que implementa a aplicação cliente. Pode utilizar o padrão Page caches para armazenar grandes quantidades de resultados e o padrão Service registry, para ser resiliente às mudanças de endereços de microsserviços utilizados pelos BFFs implementados.

Vantagens: Provê um *endpoint* menor e menos complexo, onde os clientes ficam desacoplados dos microsserviços, facilitando a troca destes serviços e diminuindo as mudanças no frontend. Previne a ocorrência de requisições concorrentes aos MSs e aumenta a autonomia do time de desenvolvimento de frontend. Fornece um grau de resiliência às aplicações, devido a possibilidade de isolamento de problemas no serviço BFF.

Tabela 14 – Padrões arquiteturais *Adapters* e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Adapter	legacy software integration	evolvability	+	[P7]
Aggregator	better service integration	agility	++	[G1]
Aggregator	efficient message exchange	performance	-	[G1]
Anti-corruption layer	fail isolation	availability	++	[G8]
Anti-corruption layer	legacy software integration	evolvability	+	[P12]
Anti-corruption layer	efficient message exchange	performance	-	[P13]
API gateway	decentralized endpoint	availability	-	[P9]
API gateway	legacy software integration	evolvability	+	[P9, P10]
API gateway	service extension	evolvability	+	[P9, P10]
API gateway	efficient message exchange	performance	-	[P9]
API gateway	services compatibility	portability	++	[P9, P10]
Backend for frontend	fail isolation	availability	+	[P3]
Backend for frontend	service extension	evolvability	+	[P2]
Backend for frontend	decoupled services	maintainability	+	[P1, P13]
Backend for frontend	efficient message exchange	performance	+	[P13]
Backend for frontend	services compatibility	portability	++	[P13]

Fonte: Produzido pelo autor

1.2.2 Padrões para APIs

Nesta categoria estão os padrões que se destinam aos aspectos ligados as APIs dos serviços, como evolução e documentação.

Aggressive obsolescence

Contexto: Em um ambiente de MSs, onde é necessário manter várias versões de uma API para compatibilidade com diferentes clientes.

Problema: Como reduzir o esforço necessário para manter versões obsoletas de APIs?

Solução: Anunciar aos clientes uma data de encerramento do suporte à API.

Vantagens: Permite a evolução do sistema, estabelecendo uma data para que APIs obsoletas sejam removidas.

Desvantagens: Pode prejudicar a disponibilidade de clientes, caso algum deles ainda esteja utilizando uma API obsoleta na data em que seu suporte é encerrado.

API versioning

Contexto: Uma API evolui e várias versões com melhorias são ofertadas. Em algum momento, as mudanças de uma nova versão não possuem compatibilidade com versões anteriores, prejudicando clientes que ainda utilizam estas versões.

Problema: Como um provedor de API pode indicar as capacidades atuais e a existência de possíveis incompatibilidades aos clientes?

Solução: Introduzir um indicador explícito de versão nas mensagens trocadas com os clientes. Este indicador possui números demonstrando o nível de comprometimento das alterações (p. ex. versão 2.3.1).

Vantagens: Permite que microserviços sejam atualizados sem prejudicar clientes que ainda utilizam versões anteriores da API, os quais podem trocar para novas versões gradualmente.

Desvantagens: Pode prejudicar a *agilidade* [P11] e a *manutenibilidade* [P11], devido à necessidade de desenvolvimento e manutenção de várias versões de um mesmo serviço.

API description

Contexto: Um provedor de serviços expõe uma ou mais operações em uma API, no entanto, sem documentar de que forma essas operações podem ser utilizadas.

Problema: Quais informações devem ser compartilhadas entre o provedor da API e seus clientes e de que forma?

Solução: Criar uma descrição da API que define as estruturas de mensagens de requisição e resposta, erros e conhecimentos técnicos relevantes para os clientes.

Vantagens: Comunica claramente aos clientes informações sobre operações, mensagens e erros, evitando ambiguidades de interpretação. Clientes sabem o que esperar das APIs e sobre sua evolução através do tempo.

Desvantagens: Pode aumentar os esforços de manutenção, devido a necessidade de constante atualização da descrição das APIs.

API Documentation and Management

Contexto: No desenvolvimento de MSs, é necessário especificar APIs para realização de testes.

Problema: Como reduzir a complexidade de geração e gerenciamento das especifica-

ções de APIs para realização de testes?

Solução: Utilizar descrições de APIs para prover uma visão geral das funcionalidades dos microsserviços. Estas descrições são atualizadas conforme os serviços evoluem, facilitando o gerenciamento das APIs.

Vantagens: Favorece a testabilidade dos MSs, fornecendo informações para que desenvolvedores possam implementar testes, como por exemplo, testes de contrato.

Consumer-driven contracts

Contexto: MSs precisam ser alterados para atender novos requisitos de seus clientes. Tais alterações levam a criação de novas interfaces específicas para cada tipo de cliente.

Problema: Como minimizar os impactos causados por alterações de interface e como saber de antemão quais clientes serão afetados por elas?

Solução: Implementar contratos entre serviços e seus clientes. Cada contrato possui uma especificação das funções que o serviço deve fornecer ao cliente. O serviço deve conhecer todos os contratos existentes. Dessa forma, quando uma alteração é feita no serviço, é possível saber quais clientes serão impactados, ou se convém desenvolver uma nova interface específica, para que clientes existentes não sejam impactados.

Vantagens: Favorece a evolução dos MSs, pois permite rastrear os impactos das alterações feitas nos serviços, minimizando seus efeitos sobre os clientes dos serviços. Permite fazer testes nos serviços sem a necessidade de participação dos clientes.

Embedded entity

Contexto: A informação requerida por um cliente de um serviço contém dados estruturados, cujos elementos se relacionam uns com os outros. Por exemplo, dados cadastrais de um consumidor podem estar relacionados com números de telefone, endereços, outros dados agregados etc. O cliente pode requerer que todos estes dados sejam retornados.

Problema: Como evitar que múltiplas requisições sejam enviadas à API para obter os diversos dados relacionados?

Solução: Fazer com que todas as informações relevantes sejam embutidas no retorno da requisição feita a API. Por exemplo, caso a API retorne os telefones de um consumidor, deve-se retornar também seus dados cadastrais.

Vantagens: Para retornar um dado e suas relações, menos requisições à API são necessárias, trazendo um ganho de performance, já que os dados estarão disponíveis mais rapidamente. No entanto, caso a quantidade de dados seja muito grande, pode demorar mais tempo para que sejam transferidos do serviço para o cliente.

Experimental preview

Contexto: No desenvolvimento de uma API, o provedor de serviços necessita liberar

o seu uso de forma experimental, para que clientes possam iniciar a implementação de integrações. Mesmo assim, o provedor gostaria de poder realizar modificações na API de forma livre.

Problema: Como a API pode ser introduzida de forma menos arriscada e obtendo um feedback dos clientes?

Solução: Liberar acesso a API sem comprometimento com as funcionalidades oferecidas, estabilidade e longevidade, deixando isso claro para os clientes.

Vantagens: Clientes podem ter acesso prévio às APIs e influenciar no seu desenvolvimento, ao passo que provedores tem flexibilidade para responder rapidamente às mudanças indicadas.

Limited lifetime guarantee

Contexto: Um provedor de serviços publica uma API que é utilizada por clientes e pretende não fazer modificações que prejudiquem seus clientes, no entanto, deseja que em algum momento possa alterar sua API.

Problema: Como um provedor pode fazer com que seus clientes saibam até quando uma API será suportada?

Solução: Cada nova versão de uma API deve ser anotada com uma data limite para suporte. A partir desta data, o provedor poderá realizar alterações na API.

Vantagens: Facilita a evolução dos microserviços, pois o suporte a versões antigas tem uma data específica para ser encerrado, favorecendo o desenvolvimento de versões desenvolvidas com tecnologias novas, sem necessidade de manter compatibilidade com APIs anteriores.

Linked information holder

Contexto: A informação requerida por um cliente de um serviço contém dados estruturados, cujos elementos se relacionam uns com os outros. Por exemplo, dados cadastrais de um consumidor podem estar relacionados com números de telefone, endereços, outros dados agregados etc. O cliente pode requerer que todos estes dados sejam retornados.

Problema: Como evitar a transferência de grandes mensagens contendo diversos dados que nem sempre são requeridos pelo cliente?

Solução: Embutir, no retorno da requisição feita à API, links para todas as informações relacionadas. Por exemplo, caso a API retorne os telefones de um consumidor, deve-se retornar apenas um link para um endpoint que contenha os dados cadastrais. Dessa forma, caso o cliente do serviço necessite destes dados, é possível acessá-los.

Vantagens: A mensagem de retorno enviada pelo serviço é menor e utiliza menos recursos de comunicação, no entanto, são necessárias mais trocas de mensagens caso o

cliente necessite de outros dados relacionados.

Desvantagens: Existe um esforço e custo maior de desenvolvimento, devido a necessidade de se implementar endpoints para retorno das informações relacionadas.

Semantic versioning

Contexto: Quando se utiliza o padrão “API versioning” para versionamento das APIs, pode não ficar claro, ao observar o número da versão, o quanto significantes são as alterações entre as diferentes versões.

Problema: Como os stakeholders podem comparar versões de APIs e detectar de forma imediata se são compatíveis?

Solução: Introduzir um esquema de versionamento de três números, x.y.z, permitindo ao provedor de uma API especificar diferentes níveis de alteração através da composição dos números identificadores, normalmente versões major, minor e patch.

Vantagens: Favorece a evolução das APIs, permitindo que novas versões sejam disponibilizadas e seus impactos nos clientes sejam percebidos de forma clara através dos níveis de alteração indicados no versionamento.

Two in production

Contexto: APIs evoluem e novas versões são melhoradas e oferecidas aos clientes. Em algum momento, as mudanças da nova versão não são mais compatíveis com versões anteriores, dessa forma prejudicando clientes que ainda não migraram para as versões mais novas.

Problema: Como um provedor pode gradualmente atualizar suas APIs sem prejudicar seus clientes, mas sem ter que manter um grande número de versões em produção?

Solução: Implantar e dar suporte a duas versões de uma API que provêm uma mesma funcionalidade, mas que não são compatíveis entre si. Dessa forma, clientes terão tempo para se adaptarem e migrarem para as novas versões.

Vantagens: Permite que microserviços evoluam sem a necessidade de ficarem atrelados a versões anteriores. APIs obsoletas são gradualmente substituídas pelas novas versões.

Tabela 15 – Padrões arquiteturais para APIs e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Aggressive obsolescence	support older API versions	availability	-	[P5]
Aggressive obsolescence	API evolution	evolvability	+	[P5]
API description	API documentation	evolvability	++	[P5]
API description	documented API	maintainability	-	[P5]
Continua na próxima página				

Tabela 15 – continuado da página anterior

Padrão	RNF impactado	RNF Pai	Impacto	Referências
API documentation and management	documented API	maintainability	+	[P4]
API versioning	time to market reduction	agility	-	[P11]
API versioning	API versioning	evolvability	++	[P11]
API versioning	documented API	maintainability	-	[P11]
Consumer-driven contracts	API evolution	evolvability	+	[P2]
Consumer-driven contracts	documented API	maintainability	+	[P2]
Embedded entity	efficient message exchange	performance	+	[P14]
Experimental preview	API evolution	evolvability	+	[P5]
Limited lifetime guaranteee	API evolution	evolvability	+	[P5]
Linked information holder	Cost reduction	agility	-	[P14]
Linked information holder	efficient message exchange	performance	+	[P14]
Semantic versioning	API versioning	evolvability	+	[P5]
Two in production	API evolution	evolvability	+	[P5]

Fonte: Produzido pelo autor

1.2.3 Padrões para Automação

Nesta categoria estão os padrões que se destinam a automação de algum aspecto ligado aos serviços.

Automated configuration

Contexto: Em uma arquitetura de MSs, a configuração das instâncias, serviços e hosts é feita manualmente pelos desenvolvedores.

Problema: Para cada MS, são feitas diferentes configurações para vários ambientes (desenvolvimento, teste e produção). A configuração manual leva mais tempo para ser realizada, sendo sujeita aos erros. Como maximizar o tempo gasto no processo de configuração e evitar erros?

Solução: Utilização de servidores e serviços para automatizar o processo de configuração, através do uso de ferramentas de gerenciamento.

Vantagens: A automatização permite aos desenvolvedores gerenciar as configurações de forma centralizada, facilitando o compartilhamento de elementos comuns, mantendo a consistência de configuração entre diferentes serviços.

Desvantagens: A configuração manual torna mais fácil o desenvolvimento de MSs, no entanto, conforme os sistemas crescem, a automatização de configuração é necessária. Isso traz uma maior complexidade de desenvolvimento, devido a introdução de novas ferramentas e monitoramento para verificar se as configurações de cada serviço estão corretas.

Automating test procedure

Contexto: No desenvolvimento de MSs, devem ser aplicados testes de integração entre serviços.

Problema: O processo manual de testes consome muito tempo, devido ao grande número de testes e quantidade de checagens de resultados. Como limitar a complexidade dos processos de testes?

Solução: Aplicação de procedimentos automáticos de teste, através do uso de ferramentas.

Vantagens: A utilização de ferramentas para automatização de testes promove a testabilidade dos MSs.

Desvantagens: A aplicação das ferramentas exige uma curva mínima de aprendizado, além de depender da geração de dados de monitoramento de MSs.

CI/CD

Contexto: A implantação independente possibilita processos de desenvolvimento e entrega contínua dos MS.

Problema: Como automatizar o gerenciamento dos processos de teste e implantação?

Solução: Utilizar ferramentas de integração e implantação para automatizar os processos de teste e implantação.

Vantagens: A integração do desenvolvimento e operação (DevOps), associado à entrega contínua, permite diminuir o tempo entre releases do sistema, mantendo a qualidade do software. Falhas no sistema podem ser identificadas de forma mais fácil. Testes e implantação se tornam mais eficientes devido à automação.

Desvantagens: Existe um aumento de esforço, tempo e custo necessário para adoção das ferramentas de CI/CD, além de que pode haver uma resistência organizacional.

Tabela 16 – Padrões arquiteturais para automação e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Automated configuration	ease of configuration	manageability	++	[P11]
Automating test procedure	automated test	maintainability	++	[P4]
CI/CD	automated deploy	agility	++	[P11]
CI/CD	automated test	agility	++	[P11]
CI/CD	automated deploy	deployability	++	[P11]
CI/CD	automated test	maintainability	++	[P6, P11, P13]
CI/CD	automated deploy	manageability	++	[P6, P11, P13]
CI/CD	automated test	manageability	++	[P6, P11, P13]

Fonte: Produzido pelo autor

1.2.4 Padrões para Balanceamento de Carga

Nesta categoria estão os padrões que se destinam ao balanceamento da carga de requisições enviadas aos serviços.

Centralized load balancing

Contexto: Várias instâncias de MSs estão sendo executadas de forma independente. As requisições devem ser distribuídas entre essas instâncias de forma a maximizar a utilização dos serviços.

Problema: Como fazer a distribuição das requisições entre as várias instâncias para que nenhum deles fique significativamente sobrecarregado comparado aos outros?

Solução: Utilizar um servidor central para controle da carga de requisições enviadas aos MSs. A distribuição das requisições é baseada pela aplicação de algoritmos, como, por exemplo, Round-Robin. O servidor também pode verificar se algum serviço não está respondendo, reencaminhando as requisições para os que estão funcionando.

Vantagens: A utilização de balanceamento traz um aumento de performance, pois as requisições podem ser encaminhadas em paralelo para vários serviços, minimizando a sobrecarga e maximizando o tempo de resposta.

Distributed load balancing

Contexto: Várias instâncias de MSs estão sendo executadas de forma independente. As requisições devem ser distribuídas entre essas instâncias de forma a maximizar a utilização dos serviços.

Problema: Como fazer a distribuição das requisições entre as várias instâncias para que nenhum deles fique significativamente sobrecarregado comparado aos outros?

Solução: Nesta solução, o balanceamento é feito pelo próprio cliente que necessita enviar uma requisição aos serviços. Este cliente deve manter uma lista de serviços, através da utilização de um serviço de descoberta de serviços. Dessa forma, o cliente sabe quem são os serviços disponíveis. Através da aplicação de algoritmos de balanceamento, o cliente pode decidir para qual instância uma requisição deve ser enviada.

Vantagens: A utilização de balanceamento traz um aumento de performance, pois as requisições podem ser encaminhadas em paralelo para vários serviços, minimizando a sobrecarga e maximizando o tempo de resposta.

Tabela 17 – Padrões arquiteturais para平衡amento de carga e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Centralized load balancing	load balancing	availability	++	[P6, P13]
Centralized load balancing	load balancing	performance	++	[P4]

Continua na próxima página

Tabela 17 – continuado da página anterior

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Distributed load balancing	load balancing	availability	++	[P6, P13]
Distributed load balancing	load balancing	performance	++	[P4]

Fonte: Produzido pelo autor

1.2.5 Padrões para Bancos de Dados

Nesta categoria estão os padrões que tratam sobre formas de armazenar, gravar e ler os dados manipulados pelos serviços.

Command and Query Responsibility Separation

Contexto: Em um sistema em MSs, cada serviço possui seu próprio banco de dados. Dessa forma, fazer consultas que envolvam dados de múltiplos serviços se torna mais complicado.

Problema: Como implementar consultas que retornem dados de múltiplos serviços?

Solução: Separar as operações de escrita e atualização de dados. São utilizados comandos para atualizar dados e queries para ler dados. A realização de consultas deve ser feita em uma visão que possui uma réplica dos dados de todos os serviços. Esta réplica é somente para leitura e é atualizada por um outro serviço, que se inscreve para receber notificações a cada alteração de dados feita pelos serviços.

Vantagens: Possibilita um escalonamento independente das cargas de leitura e escrita. A performance pode ser melhorada devido ao uso de esquemas de dados otimizados individualmente para leitura e escrita. A segurança é melhorada devido a facilidade de controlar entidades que fazem escrita nos dados. A separação de operações permite uma melhor modularização e manutenibilidade do sistema.

Database cluster

Contexto: MSs precisam persistir dados em um banco de dados.

Problema: Como construir uma arquitetura de banco de dados em MSs?

Solução: Armazenar dados em um cluster de banco de dados, que pode ser acessado por todos os MSs.

Vantagens: Aumenta a escalabilidade do sistema, permitindo que os bancos de dados sejam alocados para equipamentos dedicados. Este padrão é indicado para implementações com grande quantidade de tráfego de dados. Para manter a consistência dos dados, cada MS deve ter acesso a somente um subconjunto de tabelas pertinentes ao seu domínio de negócio /citeID4.

Desvantagens: Pode levar a maiores dificuldades na gerenciabilidade [P9] e aumentar a complexidade de desenvolvimento [P9] inerente a uma arquitetura de *cluster*. A disponibilidade [P9] pode ser afetada devido ao risco de falhas causado pela inclusão de um outro componente distribuído.

Database per service

Contexto: MSs precisam persistir dados em um banco de dados.

Problema: Como construir uma arquitetura de banco de dados em MSs?

Solução: Manter um banco de dados privado para cada MS e acessível somente através de sua API. As transações de cada serviço devem envolver somente seu próprio banco de dados.

Vantagens: Um banco de dados pode ser facilmente escalado em um cluster em um segundo momento, caso necessário. Times de desenvolvimento podem trabalhar independentemente em cada serviço, alterando esquemas de bancos de dados sem afetar outros times. O acesso aos dados e esquemas por parte de outros MSs é impedida, melhorando a segurança e consistência dos dados /citeID4. MSs podem utilizar tecnologias de persistência distintas /citeID5.

Desvantagens: A *performance* [P11] pode diminuir em situações em que é necessário implementar consultas que envolvem múltiplos MSs. A garantia da consistência dos dados entre as diferentes bases de dados também fica mais difícil, podendo afetar a *confiabilidade* [P11]. A gerência de várias bases de dados também pode prejudicar a *gerenciabilidade* [P11].

Event-driven

Contexto: Em um sistema em MSs, cada serviço possui seu próprio banco de dados. No entanto, algumas transações envolvem mais de um serviço e é necessário garantir a consistência entre os dados armazenados por cada serviço.

Problema: Como manter a consistência de dados entre serviços?

Solução: Implementar uma solução baseada em eventos, onde cada serviço publica um evento sempre que um dado é atualizado. Outros serviços podem se inscrever neste evento. Quando um evento for recebido, os serviços inscritos serão notificados para também atualizarem seus dados.

Vantagens: Permite manter a consistência entre os dados através de múltiplos serviços sem necessidade de utilizar transações distribuídas.

Inconsistency handler

Contexto: Para alcançar a disponibilidade, instâncias de um mesmo MS são executadas de forma distribuída. Estes serviços interagem em paralelo com um repositório

de dados, utilizando a consistência eventual em detrimento da instantânea, onde uma atualização de dados é feita em somente uma das réplicas. Após a transação ter sido finalizada com sucesso, a alteração deve ser sincronizada com as demais. Durante o período em que a alteração não foi sincronizada, pode haver uma inconsistência eventual, onde um cliente que fizer uma consulta por aquele dado pode obter um resultado ainda não atualizado.

Problema: Como manter a disponibilidade e minimizar possíveis problemas de inconsistência dos dados?

Solução: Utilizar ferramentas que realizem a sincronização dos dados através do gerenciamento das mudanças de estado realizadas nos MSs. Os eventos são capturados em logs e então replicados aos demais MSs.

Vantagens: Permite manter a disponibilidade dos MSs através da distributividade, tratando os possíveis problemas de consistência dos dados replicados entre as instâncias.

Local database proxy

Contexto: MSs precisam persistir dados em um banco de dados.

Problema: Como construir uma arquitetura de banco de dados em MSs?

Solução: Utilizar replicação entre as bases de dados através de um esquema master/slave e um proxy para direcionar as requisições. Operações de escrita são gerenciadas pelo master e replicadas para os nós slave, enquanto que as leituras são processadas pelos slaves. Cada MS deve utilizar um proxy local, o qual distribui a carga de trabalho entre master e slaves.

Vantagens: A utilização do esquema master/slave permite incluir e remover nós, provendo elasticidade em tempo de execução. A utilização de cache local e proxy para redirecionamento conforme a carga, melhora a performance dos serviços.

Desvantagens: Possui limitações quando é necessário escalar para operações de escrita.

Local sharding-based router

Contexto: MSs precisam persistir dados em um banco de dados.

Problema: Como construir uma arquitetura de banco de dados em MSs?

Solução: Dividir os dados em múltiplos bancos de dados, separando-os em grupos funcionais independentes, de forma a evitar operações join com outros bancos de dados. As requisições são processadas por um redirecionador local, que determina qual banco de dados é o mais adequado.

Vantagens: O sistema pode ser escalado através da inclusão de novos nós. O agrupamento funcional permite a escalabilidade tanto para operações de leitura quanto

de escrita. A utilização de proxy para redirecionamento conforme a carga, associado ao balanceamento entre os fragmentos, melhora a performance dos serviços.

Shared database server

Contexto: MSs precisam persistir dados em um banco de dados.

Problema: Como construir uma arquitetura de banco de dados em MSs?

Solução: Armazenar dados em um único banco de dados, que pode ser acessado por todos os MSs.

Vantagens: Útil em situações de migração de sistema monolítico para MS, permitindo que o esquema de dados seja reutilizado na nova arquitetura sem grandes modificações.

Desvantagens: A independência dos MSs é quebrada, pois todos os MSs do sistema podem acessar os mesmos bancos de dados.

Tabela 18 – Padrões arquiteturais para bancos de dados e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Command and query responsibility separation	isolated database	independency	+	[P13]
Command and query responsibility separation	database segregation	maintainability	+	[P13]
Command and query responsibility separation	efficient database query	performance	+	[P13]
Command and query responsibility separation	data security	security	+	[P13]
Database cluster	easier database management	agility	-	[P9]
Database cluster	database availability	availability	-	[P9]
Database cluster	easier database management	manageability	-	[P9]
Database cluster	efficient database query	performance	+	[P10]
Database cluster	database scalability	scalability	++	[P6, P10]
Database per service	database availability	availability	-	[P11]
Database per service	database evolution	evolvability	+	[P11]
Database per service	isolated database	independency	++	[P10, P11]
Database per service	database segregation	maintainability	++	[P11]
Database per service	easier database management	manageability	-	[P11]
Database per service	efficient database query	performance	-	[P11]
Database per service	database scalability	scalability	++	[P6, P10]
Database per service	data security	security	+	[P11]
Event-driven	database availability	availability	+	[G17]
Inconsistency handler	database availability	availability	+	[P4]
Local database proxy	efficient database query	performance	+	[G7]
Local database proxy	database scalability	scalability	++	[G7]
Local sharding-based router	efficient database query	performance	+	[G7]
Local sharding-based router	database scalability	scalability	++	[G7]
Shared database server	legacy software migration	evolvability	+	[P9, P10]

Continua na próxima página

Tabela 18 – continuado da página anterior

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Shared database server	isolated database	independency	–	[?]

Fonte: Produzido pelo autor

1.2.6 Padrões para Cache de Resultados

Neste categoria estão os padrões utilizados para implementação de algum tipo de cache nos resultados processados e retornados pelos serviços, visando otimizar o processamento e tráfego de conteúdo na comunicação entre os serviços.

Page cache

Contexto: Um sistema em MSs utiliza o padrão backend for frontend para construção de interfaces customizadas para clientes (p. ex., web ou mobile).

Problema: Um MS que representa uma entidade de negócios pode retornar informação em excesso, a qual não pode ser facilmente exibida, principalmente em aplicações mobile.

Solução: Utilizar o padrão page cache em conjunto com o padrão backend for frontend. Uma interface específica deve ser disponibilizada ao cliente, para que ele possa solicitar um conjunto limitado de uma coleção muito maior de dados. Os dados ficam armazenados em cache no serviço do BFF e podem ser solicitados pelo cliente, que vai consumindo-os conforme necessário.

Vantagens: Existe um ganho de performance do ponto de vista do cliente, pois menos tráfego é transportado pela rede. O cliente não precisa esperar o retorno da consulta do grande volume de dados, pois eles já estarão em cache no serviço do BFF. Além disso, menos dados chegam ao cliente e podem ser mais rapidamente exibidos.

Results cache

Contexto: Fazer chamadas a serviços remotos pode ser custoso, devido a latência e tempo de processamento nos serviços.

Problema: Como melhorar a performance quando é necessário fazer repetidas chamadas aos serviços?

Solução: Utilizar um cache local de resultados das chamadas, diminuindo a necessidade de chamar várias vezes um serviço remoto. Para que os dados não se tornem obsoletos, deve-se usar um tempo de cache baixo.

Vantagens: Melhora a performance do serviço que faz a chamada a outro serviço remoto, diminuindo o número de requisições, permitindo que as operações sejam feitas

com os dados armazenados localmente.

Scalable store

Contexto: Em um sistema baseado em MSs, é necessário persistir estados para representar as interações anteriores e atuais dos usuários.

Problema: Como representar a persistência de estados em uma aplicação de MSs?

Solução: Persistir os estados em uma área de armazenamento escalável, disponível e compartilhada por qualquer número de aplicações. Pode-se utilizar, por exemplo, bancos de dados relacionais ou “NoSQL”. Os padrões results cache e pages cache geralmente utilizam o padrão Scalable store.

Vantagens: Permite a escalabilidade da área de armazenamento compartilhada, desta forma aumentando a disponibilidade dos serviços.

Tabela 19 – Padrões arquiteturais para cache de resultados e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Page cache	efficient message exchange	performance	+	

Fonte: Produzido pelo autor

1.2.7 Padrões para Configuração

Nesta categoria estão os padrões que tratam sobre a forma de de tratamento das configurações dos serviços.

Externalized configuration

Contexto: Aplicações normalmente utilizam vários serviços externos, como processamento de pagamentos, e-mails, mensagens etc.

Problema: Como fazer com que serviços sejam executados em diferentes ambientes (p. ex., desenvolvimento, teste, produção) sem modificação?

Solução: Externalizar todas as configurações da aplicação, incluindo credenciais de conexão a banco de dados e endereços de rede. Na inicialização de um serviço, as configurações são lidas de uma fonte externa, como variáveis de ambiente ou arquivos de configuração.

Vantagens: A aplicação pode rodar em múltiplos ambientes sem modificações ou recompilações. A segurança é melhorada, pois o acesso às configurações fica externo aos serviços e pode ser restringido.

Tabela 20 – Padrões arquiteturais para configuração e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Externalized configuration	configuration portability	portability	++	[G18]
Externalized configuration	configuration security	security	+	[P12, P13]

Fonte: Produzido pelo autor

1.2.8 Padrões para Decomposição dos Serviços

Nesta categoria estão os padrões que tratam sobre a decomposição de sistemas maiores em pequenos serviços coesos e com baixo acoplamento entre si.

Contributor Coupling

Contexto: Na migração de um sistema monolítico para MS, o sistema deve ser decomposto em pequenos serviços, buscando-se a correta granularidade dos serviços, objetivando uma alta coesão e baixo acoplamento.

Problema: Como obter uma alta coesão e baixo acoplamento na decomposição de sistemas monolíticos para MSs?

Solução: Fazer a divisão dos serviços baseando-se em aspectos dos times de desenvolvimento. Por exemplo, módulos do sistema monolítico que giram em torno das mesmas capacidades de negócio e que são desenvolvidos por um mesmo grupo de pessoas poderiam ser colocados em um mesmo MS, dessa forma diminuindo a necessidade de comunicação externa e maximizando a comunicação interna e a coesão dentro do time de desenvolvimento.

Vantagens: A correta decomposição dos MSs melhora a escalabilidade do sistema, a independência dos serviços e a sua implantação, dando mais agilidade ao processo de desenvolvimento. A manutenibilidade é melhorada devido à coesão dos serviços e ao baixo acoplamento.

Decompose by business capabilities

Contexto: Na migração de um sistema monolítico para MS, o sistema deve ser decomposto em pequenos serviços, buscando-se a correta granularidade dos serviços, objetivando uma alta coesão e baixo acoplamento.

Problema: Como obter uma alta coesão e baixo acoplamento na decomposição de sistemas monolíticos para MSs?

Solução: Definir os serviços de acordo com as capacidades de negócio. Uma capacidade de negócio é algo que organização faz e que gera valor, como, por exemplo, gerenciamento de compras ou gerenciamento de clientes. Cada capacidade de negócio

corresponde a um serviço.

Vantagens: Os serviços tendem a ficarem estáveis, pois as capacidades de negócio também são estáveis. Times de desenvolvimento organizados ao redor dos valores de negócio ao invés de características técnicas. A manutenibilidade é melhorada devido à coesão dos serviços e ao baixo acoplamento.

Decompose by subdomain

Contexto: Na migração de um sistema monolítico para MS, o sistema deve ser decomposto em pequenos serviços, buscando-se a correta granularidade dos serviços, objetivando uma alta coesão e baixo acoplamento.

Problema: Como obter uma alta coesão e baixo acoplamento na decomposição de sistemas monolíticos para MSs?

Solução: Definir os serviços de forma que correspondam aos subdomínios identificados na análise feita através do DDD (Domain-Driven Design). No DDD, o domínio é o negócio da organização e consiste de múltiplos subdomínios, cada qual correspondendo a diferentes partes do negócio.

Vantagens: Os serviços tendem a ficarem estáveis, pois os subdomínios também são estáveis. Times de desenvolvimento organizados ao redor dos valores de negócio ao invés de características técnicas. A manutenibilidade é melhorada devido à coesão dos serviços e ao baixo acoplamento.

Logical coupling

Contexto: Na migração de um sistema monolítico para MS, o sistema deve ser decomposto em pequenos serviços, buscando-se a correta granularidade dos serviços, objetivando uma alta coesão e baixo acoplamento.

Problema: Como obter uma alta coesão e baixo acoplamento na decomposição de sistemas monolíticos para MSs?

Solução: Fazer a divisão dos serviços baseando-se nos princípios da responsabilidade única dos MSs, ou seja, elementos de software que são alterados pela mesma razão devem ser mantidos unidos. Dessa forma, os limites de cada serviço são determinados pelas suas responsabilidades ou pelas suas mudanças de comportamento.

Vantagens: A correta decomposição dos MSs melhora a escalabilidade do sistema, a independência dos serviços e a sua implantação, dando mais agilidade ao processo de desenvolvimento. A manutenibilidade é melhorada devido à coesão dos serviços e ao baixo acoplamento.

Semantic coupling

Contexto: Na migração de um sistema monolítico para MS, o sistema deve ser

decomposto em pequenos serviços, buscando-se a correta granularidade dos serviços, objetivando uma alta coesão e baixo acoplamento.

Problema: Como obter uma alta coesão e baixo acoplamento na decomposição de sistemas monolíticos para MSs?

Solução: Fazer a divisão dos serviços baseando-se nas noções de limites de contexto originárias do DDD (Domain-driven Design). Os limites de cada MS são determinados através do exame de conteúdo e semântica do código fonte do sistema.

Vantagens: A correta decomposição dos MSs melhora a escalabilidade do sistema, a independência dos serviços e a sua implantação, dando mais agilidade ao processo de desenvolvimento. A manutenibilidade é melhorada devido à coesão dos serviços e ao baixo acoplamento.

Tabela 21 – Padrões arquiteturais para decomposição dos serviços e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Contributor coupling	decomposed services	deployability	++	[P4]
Contributor coupling	decomposed services	independency	++	[P4]
Contributor coupling	decomposed services	maintainability	++	[P4]
Contributor coupling	decomposed services	scalability	++	[P4]
Decompose by business capabilities	decomposed services	deployability	++	[P4]
Decompose by business capabilities	decomposed services	independency	++	[P4]
Decompose by business capabilities	decomposed services	maintainability	++	[P4]
Decompose by business capabilities	decomposed services	scalability	++	[P4]
Decompose by subdomain	decomposed services	deployability	++	[P4]
Decompose by subdomain	decomposed services	independency	++	[P4]
Decompose by subdomain	decomposed services	maintainability	++	[P4]
Decompose by subdomain	decomposed services	scalability	++	[P4]
Logical coupling	decomposed services	deployability	++	[P4]
Logical coupling	decomposed services	independency	++	[P4]
Logical coupling	decomposed services	maintainability	++	[P4]
Logical coupling	decomposed services	scalability	++	[P4]
Semantic coupling	decomposed services	deployability	++	[P4]
Semantic coupling	decomposed services	independency	++	[P4]
Semantic coupling	decomposed services	maintainability	++	[P4]
Semantic coupling	decomposed services	scalability	++	[P4]

1.2.9 Padrões para Descoberta de Serviços

Nesta categoria estão os padrões que tratam sobre as formas que podem ser implementadas visando a descoberta de serviços.

Client-side discovery

Contexto: Múltiplas instâncias de um mesmo MS são executadas em diferentes containers. A comunicação entre eles precisa ser definida de forma dinâmica e eficiente.

Problema: Como implementar a descoberta de serviços entre os MSs?

Solução: Implementar um serviço registrador, que faz a tarefa de dinamicamente transformar endereços DNS em IPs. Para se comunicar com um serviço, o cliente primeiro faz a solicitação a um serviço registrador para descobrir o endereço do serviço. Posteriormente, o cliente faz a requisição diretamente ao serviço desejado, podendo utilizar algoritmos de balanceamento para decidir qual serviço receberá a requisição.

Vantagens: Facilita a comunicação direta entre cliente e serviço. Permite maior resiliência, através da detecção de problemas e reinício de serviços e facilita a migração, pois serviços legados podem ser substituídos simplesmente trocando-se o endereço no serviço de registros.

Server-side discovery

Contexto: Múltiplas instâncias de um mesmo MS são executadas em diferentes containers. A comunicação entre eles precisa ser definida de forma dinâmica e eficiente.

Problema: Como implementar a descoberta de serviços entre os MSs?

Solução: Implementar um serviço balanceador, responsável por receber as requisições dos clientes, e um serviço registrador, que faz a tarefa de dinamicamente transformar endereços DNS em IPs. O balanceador, ao receber uma requisição, solicita ao serviço registrador os endereços das instâncias disponíveis. O balanceador decide qual instância será utilizada para atender à requisição, devolvendo o endereço para o cliente, que se comunicará diretamente com o serviço.

Vantagens: Permite maior resiliência, através da detecção de problemas e reinício de serviços e facilita a migração, pois serviços legados podem ser substituídos simplesmente trocando-se o endereço no serviço de registros.

Tabela 22 – Padrões arquiteturais para descoberta de serviços e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Client-side discovery	load balancing	availability	++	[P9]
Client-side discovery	easier service replacement	evolvability	++	[P9]
Client-side discovery	load balancing	performance	++	[P9]

Continua na próxima página

Tabela 22 – continuado da página anterior

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Server-side discovery	load balancing	availability	++	[P9]
Server-side discovery	easier service replacement	evolvability	++	[P9]
Server-side discovery	load balancing	performance	++	[P9]

Fonte: Produzido pelo autor

1.2.10 Padrões para Escala dos Serviços

Nesta categoria estão os padrões que definem a forma como é feita a escala dos serviços em uma arquitetura de MSs.

Proactive auto-scaling

Contexto: Serviços precisam ser escalados de forma horizontal, ou seja, é necessário aumentar o número de instâncias executando um mesmo serviço.

Problema: Como escalar horizontalmente os serviços?

Solução: Utilizar uma abordagem proativa de escala, escalando o sistema baseado em algum processo de previsão de carga, como, por exemplo, o histórico de requisições para cada MS ou pela conversão das requisições em quantidade de CPU e memória utilizada.

Vantagens: Permite que o sistema seja escalado de forma automática, no entanto, é necessário que o sistema esteja rodando a algum tempo para que existam dados históricos para realizar a previsão.

Reactive auto-scaling

Contexto: Serviços precisam ser escalados de forma horizontal, ou seja, aumentar o número de instâncias executando um mesmo serviço.

Problema: Como escalar horizontalmente os serviços?

Solução: Utilizar uma abordagem reativa de escala, ou seja, monitorar métricas e, através de limiares, decidir se um MS deve ter o número de instâncias aumentado ou diminuído. Por exemplo, pode-se monitorar os valores de utilização de CPU, latência e carga de rede ou utilizar métricas de filas de mensagens. Caso um valor extrapole um limiar superior, o próprio sistema de gerenciamento deve aumentar as instâncias, ou, caso ultrapasse um limiar inferior, é feita a diminuição.

Vantagens: Permite que o sistema seja escalado de forma automática.

Tabela 23 – Padrões arquiteturais para escala dos serviços e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Proactive auto-scaling	auto-scaling	scalability	++	[P4]
Reactive auto-scaling	auto-scaling	scalability	++	[P4]

Fonte: Produzido pelo autor

1.2.11 Padrões para Implantação dos Serviços

Nesta categoria estão os padrões que definem como os serviços serão implantados.

Blue green deployment

Contexto: Em um sistema em MSs, é necessário implantar novos serviços ou versões de serviços existentes.

Problema: Como eliminar o tempo de indisponibilidade de um sistema ao implantar um serviço?

Solução: Utilizar o padrão blue green, que consiste em ter dois ambientes idênticos de produção, onde, a qualquer momento, qualquer um dos ambientes pode estar em produção. Ao realizar uma nova implantação, deve-se fazê-lo em apenas um dos ambientes, blue ou green. Após a implantação, devem ser feitos testes para garantir que tudo está funcionando, para então alterar as configurações de roteamento e encaminhar as requisições para o servidor que possui os novos serviços. Se algo inesperado acontecer, pode-se voltar a utilizar o servidor anterior, onde não foram feitas alterações nos serviços.

Vantagens: Elimina o tempo de indisponibilidade dos serviços.

Canary release

Contexto: Em um sistema em MSs, é necessário implantar novos serviços ou versões de serviços existentes.

Problema: Como reduzir os riscos de implantar uma nova versão de um serviço em um ambiente de produção?

Solução: Implantar o serviço em um subconjunto da infraestrutura e disponibilizá-lo para um grupo restrito de usuários.

Vantagens: Esta abordagem permite implantar gradualmente um serviço para poucos usuários e ir aumentando conforme se ganha confiança na performance do serviço. Caso algo errado ocorra, pode-se rotear os usuários para as versões anteriores rapidamente.

Containerization

Contexto: Em um sistema em MSs, cada serviço é implantado como um conjunto

de instâncias de serviço.

Problema: Como simplificar a implantação dos serviços?

Solução: Utilizar containers para implantação, os quais incluem tudo que é necessário para um MS ser executado, como bibliotecas e dados.

Vantagens: Apresenta uma melhor performance na comunicação com a máquina hospedeira. Permite melhor aproveitamento dos recursos disponíveis, reduzindo custos de infraestrutura. A escalabilidade das instâncias é facilitada devido a característica mais leve dos containers. O isolamento de cada instância favorece a segurança e a independência dos serviços.

Deploy into a Cluster and Orchestrate Containers

Contexto: Um sistema em MSs utiliza CI (Continuous Integration).

Problema: Como implantar instâncias de serviços em um cluster? Como orquestrar a implantação de todos os serviços com o menor esforço possível?

Solução: Utilizar um sistema para gerenciamento do cluster, onde é possível implantar containers de serviços sob demanda e gerenciar falhas e reinicialização de nós caso necessário.

Vantagens: O gerenciamento de falhas melhora a confiabilidade dos MSs. A utilização de uma ferramenta traz mais agilidade na implantação.

Multiple service per host

Contexto: Em um sistema em MSs, cada serviço é implantado como um conjunto de instâncias de serviço, objetivando aumentar o desempenho e disponibilidade.

Problema: Como empacotar e implantar os serviços?

Solução: Executar múltiplas instâncias de serviços em um mesmo servidor.

Vantagens: Facilita a escalabilidade devido a possibilidade de implantar várias instâncias em um mesmo servidor. Melhor utilização dos recursos disponíveis em cada servidor.

Desvantagens: Dificulta o monitoramento e rastreabilidade dos serviços, devido a ter várias instâncias rodando no mesmo servidor e utilizando os mesmos recursos.

Single service per host

Contexto: Em um sistema em MSs, cada serviço é implantado como um conjunto de instâncias de serviço, objetivando aumentar o desempenho e disponibilidade.

Problema: Como empacotar e implantar os serviços?

Solução: Executar cada instância de serviço em servidores separados.

Vantagens: Permite isolar completamente os serviços, reduzindo a possibilidade de conflito de recursos. Facilita o monitoramento dos recursos utilizados pelos serviços devido a cada instância executar um único serviço.

Desvantagens: Reduz a eficiência na utilização de recursos, já que é necessário mais hosts, um para cada serviço. A escalabilidade pode ser prejudicada, pois, para escalar um serviço, é necessário escalar o host inteiro, utilizando mais recursos.

Tabela 24 – Padrões arquiteturais para implantação dos serviços e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Blue green deployment	reduce service downtime when deploying	availability	++	[G6]
Canary release	reduce service downtime when deploying	availability	++	[G23]
Containerization	containerized services	availability	++	[P6, P13]
Containerization	efficient deploy strategy	deployability	++	[P6]
Containerization	isolated service instance	independency	++	[P8]
Containerization	efficient host communication	performance	++	[P4]
Containerization	Lightweight runtime environment	scalability	+	[P6]
Containerization	platform security	security	+	[P7]
Deploy into a cluster and orchestrate containers	service orchestration	agility	+	[P7]
Deploy into a cluster and orchestrate containers	fail isolation	availability	+	[P7, P12, P13]
Deploy into a cluster and orchestrate containers	orchestrated deploy	deployability	++	[P7]
Deploy into a cluster and orchestrate containers	service orchestration	manageability	++	[P7]
Multiple service per host	efficient deploy strategy	deployability	++	[G22]
Multiple service per host	resources use monitoring	monitorability	-	[G22]
Multiple service per host	Lightweight runtime environment	scalability	+	[G22]
Single service per host	efficient deploy strategy	deployability	++	[P9]
Single service per host	isolated service instance	independency	++	[P9, P10, P11]
Single service per host	resources use monitoring	monitorability	+	[G22]
Single service per host	efficient resources utilization	performance	-	[G22]
Single service per host	Lightweight runtime environment	scalability	-	[P9]

Fonte: Produzido pelo autor

1.2.12 Padrões para Integração entre Serviços

Nesta categoria estão os padrões que tratam a forma como os serviços se comunicam e se integram.

Ambassador

Contexto: MSs precisam acessar componentes compartilhados que realizam tarefas comuns, como monitoramento, log e auditoria.

Problema: Os componentes compartilhados não podem ser copiados de forma redundante para o ambiente do MS, pois são desenvolvidos de forma independente. Por outro lado, o acesso remoto a eles pode ser ineficiente.

Solução: Um container especial é criado para armazenar componentes utilitários, que farão a comunicação entre o MS e os componentes compartilhados, os quais são executados em instâncias remotas.

Vantagens: Melhora a performance no acesso aos componentes remotos, através da utilização dos componentes compartilhados que são executados no mesmo host que o serviço. Pode auxiliar na manutenibilidade, criando uma interface entre MSs e componentes compartilhados, os quais são mantidos por outras equipes, além de permitir isolar falhas, ajudando na resiliência dos MSs.

Asynchronous messaging

Contexto: MSs precisam lidar com requisições recebidas de seus clientes e de outros serviços.

Problema: Como fazer a comunicação entre serviços em uma arquitetura de MSs?

Solução: Utilizar troca de mensagens assíncrona para comunicação entre serviços, através de canais de mensagem (message brokers).

Vantagens: Baixo acoplamento dos serviços em tempo de execução e aumento da disponibilidade dos serviços, devido aos message brokers fazerem buffer de mensagens até que o recebedor possa consumi-las.

Change Code Dependency to Service Call

Contexto: Em um sistema em MSs, existe um componente que está atuando como uma dependência para outros serviços ou componentes;

Problema: Quando é apropriado alterar a dependência a nível de código para a dependência a nível de serviço?

Solução: Deve-se tentar manter o código dos serviços o mais separado possível. Quando códigos-fonte são compartilhados, existe uma chance maior de falha nos componentes dependentes.

Vantagens: A mudança para dependência de serviço torna os MSs mais independentes, melhorando a manutenibilidade do sistema.

Competing consumers

Contexto: MSs devem lidar com um grande número de requisições. Uma forma comum para implementar o tratamento destas requisições é utilizar um sistema de mensagens, onde as requisições são enviadas por este canal de comunicação até os serviços consumidores, que farão o processamento da requisição e devolverão o resultado.

Problema: Como gerenciar as requisições recebidas de forma que cada requisição seja enviada para um serviço consumidor específico e de forma balanceada, evitando a sobrecarga dos consumidores?

Solução: Utilizar uma fila de mensagens para implementar um canal de comunicação entre a aplicação e as instâncias do serviço consumidor. As requisições são inseridas na fila e processadas pelos consumidores.

Vantagens: A confiabilidade é melhorada pois, caso ocorra uma falha de um serviço consumidor, a requisição retorna para a fila e pode ser processada por outra instância que esteja em funcionamento. Os serviços consumidores podem ser escalados dinamicamente conforme a carga de processamento requerida.

Edge server

Contexto: Um sistema legado foi decomposto em MSs.

Problema: Como esconder dos sistemas clientes a complexidade da estrutura interna dos MSs? Como o status e uso dos serviços pode ser monitorado?

Solução: Através da inclusão de uma nova camada ao sistema, atuando como uma porta de entrada para os serviços. Esta camada é responsável por fazer um roteamento dinâmico das requisições recebidas aos MSs disponíveis. Dessa forma, mudanças internas na estrutura dos serviços não afetam os clientes, sendo possível também interceptar e monitorar todo o tráfego. Para evitar que a nova camada seja um ponto único de falha, deve ser utilizado mecanismos de平衡amento de carga.

Vantagens: Permite que a estrutura dos serviços seja alterada sem causar tanto impacto nos clientes. Por ser um ponto único de entrada, facilita a monitorabilidade dos MSs.

Gateway offloading

Contexto: Em um sistema de MSs, funcionalidades são utilizadas por múltiplos serviços e requerem configuração, gerenciamento e manutenção. Serviços que são distribuídos com várias aplicações aumentam a carga de gerenciamento. Alterações na funcionalidade compartilhada precisam ser implantadas em todos os serviços que compartilham a função.

Problema: Como minimizar os problemas causados pelo compartilhamento de funcionalidades?

Solução: Mover as funcionalidades compartilhadas para um gateway, liberando

assim os serviços, que podem focar apenas nas suas tarefas de negócio.

Vantagens: Melhora a manutenibilidade, pois desacopla funcionalidades e permite que times especializados possam desenvolvê-las.

Gateway routing

Contexto: Em um sistema de MSs, um cliente precisa consumir múltiplos serviços, os quais possuem endereços de rede diferentes e que devem ser gerenciados. Caso a API de um MS seja alterada, todos os clientes que a utilizam deverão ser alterados.

Problema: Como diminuir a necessidade de gerenciamento de vários endereços por parte dos clientes e minimizar impactos que mudanças em APIs podem causar nos clientes?

Solução: Inserir um gateway na frente de um conjunto de MSs. Este gateway é responsável por receber as requisições dos clientes, encaminhá-las para os serviços e então devolver aos clientes o resultado.

Vantagens: O cliente, ao invés de gerenciar vários endereços de serviços, deverá gerenciar apenas o endereço do gateway. Impactos nos clientes, causados por alterações na API dos serviços, podem ser minimizados, pois o gateway pode manter uma API simplificada, enquanto o serviço pode evoluir para interfaces mais complexas do mesmo serviço.

Desvantagens: O gateway pode inserir um ponto único de falha, além de poder se tornar um gargalo na comunicação do cliente com os serviços.

Hybrid

Contexto: Em um ambiente de MSs, é necessário haver comunicação entre os MSs.

Problema: Como implementar a comunicação entre MSs?

Solução: Implementar uma combinação entre os padrões Service Registry e API Gateway, substituindo este último por um barramento para comunicação entre MSs. Clientes podem se comunicar com o barramento, o qual atua como um registrador de serviços, direcionando as requisições para os MSs.

Vantagens: Facilita a migração de sistemas em SOA, os quais já utilizam barramento para comunicação entre serviços.

Desvantagens: Um barramento único para comunicação pode afetar a disponibilidade de todo sistema, caso ocorra uma falha neste ponto de comunicação.

Pipes and filters

Contexto: Uma aplicação deve executar várias tarefas de complexidades diferentes sobre uma informação de seu domínio. Uma abordagem seria implementar esse processamento em um módulo monolítico, no entanto, reduziria as oportunidades de refatoramento

de código, otimização e reuso. Além disso, algumas tarefas são diferentes das outras, requerendo maior ou menor capacidade computacional.

Problema: Como implementar esse tipo de processamento de forma a maximizar as oportunidades de refatoramento de código, otimização e reuso?

Solução: Decompor as atividades de processamento em componentes (ou filtros), cada um executando uma única tarefa. Através da padronização do formato de dados que cada filtro recebe e envia, eles podem ser combinados em uma sequência de processamento (pipeline).

Vantagens: Evita a duplicação de código e facilita a inclusão e remoção de componentes. Filtros podem ser escalados e executados em paralelo, dando mais poder de processamento para componentes que mais necessitam, melhorando a performance. Caso um filtro falhe, o processamento pode ser reagendado ou então outra instância do filtro pode executar o processamento, aumentando a disponibilidade do sistema.

Desvantagens: Sua implementação pode trazer mais complexidade, principalmente se os filtros estiverem distribuídos em diferentes servidores.

Priority queue

Contexto: Aplicações podem delegar tarefas específicas para outros serviços. Para isso, são utilizadas filas de mensagens que devem ser processadas em segundo plano. Normalmente, a ordem em que essas mensagens são recebidas não é importante, mas em outros casos é necessário priorizar requisições específicas.

Problema: Como priorizar requisições em uma fila de mensagens?

Solução: Utilizar uma fila de mensagens com atribuição de prioridade, para que a ordenação das mensagens a serem processadas seja feita pela ordem de prioridade e não pela ordem de chegada.

Vantagens: Permite priorizar disponibilidade e performance para clientes específicos, permitindo oferecer níveis diferentes de serviços conforme a necessidade.

Sidecar

Contexto: Um MS precisa ter acesso a componentes e serviços que proveem tarefas comuns, como monitoramento, log e auditoria. A cópia destes componentes em cada MS se torna redundante, ao mesmo tempo que o acesso remoto a eles se torna ineficiente.

Problema: Como fazer com que MSs interajam eficientemente com componentes e serviços utilitários?

Solução: Componentes e serviços utilitários devem ser implantados no mesmo host que o MS, mas dentro de um container a parte. Este container é chamado de sidecar.

Vantagens: Melhora a performance dos MSs, pois o acesso aos componentes utilitá-

rios é feito no mesmo host.

Tabela 25 – Padrões arquiteturais para integração entre serviços e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Ambassador	fail isolation	availability	+	[P3]
Ambassador	decoupled services	maintainability	+	[P13]
Ambassador	efficient shared components communication	performance	+	[G3]
Asynchronous messaging	message broking	availability	+	[G20]
Change code dependency to service call	separated services code	independency	++	[P7]
Change code dependency to service call	decoupled services	maintainability	+	[P7]
Competing consumers	message broking	availability	+	[G9]
Competing consumers	eficient message processing	performance	+	[P12, P13]
Edge server	service extension	evolvability	+	[G4]
Edge server	communication monitoring	monitorability	++	[G4]
Gateway offloading	decoupled services	maintainability	+	[P13], [G12]
Gateway routing	descentralized endpoint	availability	-	[P3], [G13]
Gateway routing	service extension	evolvability	+	[G13]
Gateway routing	decoupled services	maintainability	+	[G13]
Gateway routing	efficient message exchange	performance	-	[G13]
Hybrid	descentralized endpoint	availability	-	[P10]
Hybrid	legacy software integration	evolvability	+	[P9, P10]
Pipes and filters	better service integration	agility	-	[G14]
Pipes and filters	message broking	availability	+	[P12, P13]
Pipes and filters	decoupled services	maintainability	+	[P12, P13]
Pipes and filters	eficient message processing	performance	+	[G14]
Priority queue	message broking	availability	+	[G15]
Priority queue	eficient message processing	performance	+	[G15]
Sidecar	efficient shared components communication	performance	+	[G2]

Fonte: Produzido pelo autor

1.2.13 Padrões para Migração

Nesta categoria estão os padrões que tratam sobre a migração de sistemas monolíticos para microsserviços.

Strangler

Contexto: À medida que os sistemas envelhecem, as tecnologias utilizadas para seu desenvolvimento e sua arquitetura tendem a se tornar obsoletas. Conforme novas funcionalidades são adicionadas, sua complexidade aumenta, tornando a manutenção ou inclusão de novas funcionalidades bastante difícil. Buscando modernizar seus sistemas,

organizações buscam substituí-los, utilizando a arquitetura de MSs. No entanto, a completa substituição do sistema legado é uma tarefa bastante complexa.

Problema: Como fazer a substituição de um sistema legado por um novo sistema desenvolvido em MS?

Solução: Fazer uma substituição incremental de funcionalidades específicas do sistema legado para MSs. Criar uma interface que intercepta requisições vindas do sistema legado e encaminha para a aplicação legada ou para os novos serviços. Os clientes podem continuar utilizando a interface, e a migração pode ocorrer de forma transparente, até que se substitua todo o sistema antigo.

Vantagens: Permite a evolucionabilidade do sistema legado, que pode ser migrado para uma nova arquitetura, além de manter a compatibilidade com o código antigo até que tenha sido migrado.

Tabela 26 – Padrões arquiteturais para migração e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Strangler	legacy software migration	evolvability	+	[P2]
Strangler	legacy code compatibility	portability	++	[P2]

Fonte: Produzido pelo autor

1.2.14 Padrões para Monitoramento e controle de falhas

Nesta categoria estão os padrões utilizados para monitoramento dos serviços e controle de possíveis falhas que possam ocorrer, minimizando o impacto causado sobre os clientes dos serviços.

Aggregation processing of monitored data

Contexto: Diferentes tipos de dados sobre os MSs foram coletados através do monitoramento dos serviços.

Problema: Como processar e analisar os dados de monitoramento coletados?

Solução: Armazenar os dados coletados em formato agregado, reduzindo, dessa forma, o espaço de armazenamento necessário.

Vantagens: Com a redução do espaço de armazenamento, os dados agregados podem ser armazenados por um longo prazo.

Application metrics

Contexto: Sistemas desenvolvidos em MSs são frequentemente sujeitos a contratos de garantia de nível de serviço. Dessa forma, é crucial o monitoramento de sua performance.

Problema: Como fazer o monitoramento da performance dos MSs?

Solução: Utilizar ferramentas de monitoramento que permitam coletar dados e estatísticas sobre performance e falhas de cada MS pertencente a um sistema.

Vantagens: O monitoramento de todos os aspectos dos MSs possibilita um entendimento completo sobre o comportamento de uma aplicação, provendo dados que permitem melhorar a performance e qualidade dos serviços.

Desvantagens: A utilização de ferramentas e infraestrutura de monitoramento pode requerer o refatoramento dos serviços e a unificação de logs de serviços heterogêneos, levando a um aumento na complexidade de desenvolvimento.

Arbitral monitor

Contexto: Em um ambiente de MSs, é necessário monitorar os serviços para detectar ou antecipar falhas, objetivando tomar ações para que o sistema se recupere.

Problema: Como fazer o monitoramento dos serviços?

Solução: Um grupo de serviços independentes é responsável por detectar falhas nos serviços. Caso algum nó detecte uma falha, ela deve ser confirmada pela maioria dos demais nós pertencentes ao grupo.

Vantagens: Através da checagem de falhas, é possível evitar que requisições sejam encaminhadas para o serviço até que ele se recupere da falha.

Bulkhead

Contexto: Uma aplicação é composta por múltiplos MSs, cada um atendendo a um ou mais clientes. Uma carga excessiva ou uma falha em um serviço irá impactar todos os consumidores deste serviço.

Problema: Como evitar que a sobrecarga ou falha de um serviço impacte todos os clientes deste serviço?

Solução: Particionar as instâncias dos MSs em diferentes grupos, baseado na carga de consumo e requisitos de disponibilidade. Caso uma falha ocorra, ela pode ser isolada, causando impacto somente nos clientes daquele grupo de serviços. Dessa forma, é possível sustentar a funcionalidade do serviço para alguns consumidores que não estejam alocados àquele grupo.

Vantagens: Melhora a disponibilidade dos serviços, permitindo isolar falhas, mantendo o serviço em funcionamento nas demais instâncias alocadas em outros grupos.

Centralized storage of monitored data

Contexto: Em um sistema desenvolvido em MSs, são coletados dados obtidos através de monitoramento dos serviços.

Problema: Como armazenar os dados coletados através do monitoramento dos MSs?

Solução: Armazenar os dados coletados de todos os MSs em um componente central de armazenamento.

Vantagens: Possibilita a análise dos logs em um lugar centralizado, diminuindo a necessidade de administração.

Desvantagens: Um único ponto central de armazenamento pode se tornar um ponto único de falha.

Centralized monitor

Contexto: Em um ambiente de MSs, é necessário monitorar os serviços para detectar ou antecipar falhas, objetivando tomar ações para que o sistema se recupere.

Problema: Como fazer o monitoramento dos serviços?

Solução: Utilização de um serviço central, o qual é responsável por coletar e atualizar informações sobre a disponibilidade dos demais serviços e atualizar o status de sua disponibilidade no registrador de serviços (service registry).

Vantagens: Através da checagem de falhas, é possível evitar que requisições sejam encaminhadas para o serviço até que ele se recupere da falha.

Desvantagens: O ponto central de monitoramento pode se tornar um ponto único de falhas.

Client-side Circuit Breaker

Contexto: Em caso de falha, MSs devem ficar indisponíveis para novas requisições.

Problema: Como evitar que MSs continuem recebendo requisições em caso de falha?

Solução: Implementar um proxy para interceptar requisições entre serviços e evitar que um serviço receba requisições caso esteja com problemas. No caso do padrão client-side, este serviço roda no mesmo servidor que o cliente dos demais MSs, interceptando todas as requisições e bloqueando as que são enviadas para serviços indisponíveis.

Vantagens: MSs podem trabalhar em conjunto, causando dependências entre eles. Este padrão minimiza efeitos em cascata causados pela falha de um dos serviços na cadeia de dependência, auxiliando na disponibilidade dos MSs.

Correlation ID

Contexto: Uma aplicação em MSs é construída usando backend for frontends e múltiplos MSs e adapters, com caminhos complexos de chamadas entre os serviços

Problema: Como fazer para depurar chamadas complexas entre MSs quando não se sabe em qual MS um problema está localizado?

Solução: Utilizar Ids de correlação, os quais geralmente são números passados em cada requisição a um serviço e encaminhados em requisições sucessivas. Quando algum serviço faz log de alguma ocorrência, este ID é logado, tornando possível identificar qual requisição causou o log e quais serviços foram executados na cadeia de atendimento da requisição.

Vantagens: Com a utilização dos Ids de correlação em associação com Log aggregator para unir os logs de todos os serviços, é possível fazer uma busca pelo ID. Através do timestamp das execuções pode-se construir o grafo de execuções, analisar chamadas e parâmetros, dessa forma facilitando o rastreamento de problemas.

Decentralized storage of monitored

Contexto: Em um sistema desenvolvido em MSs, são coletados dados obtidos através de monitoramento dos serviços.

Problema: Como armazenar os dados coletados através do monitoramento dos MSs?

Solução: Armazenar os dados coletados de forma descentralizada, localmente, em componentes localizados em cada host, plataforma ou serviço.

Vantagens: Permite uma melhor escalabilidade da solução de monitoramento, pois cada novo serviço provê um novo local de armazenamento.

Distributed logging

Contexto: Em um sistema desenvolvido em MSs, são coletados dados obtidos através de monitoramento dos serviços.

Problema: Como armazenar os dados coletados através do monitoramento dos MSs?

Solução: Cada serviço deve armazenar logs em um mesmo servidor externo, responsável por agregar estes dados.

Vantagens: Permite ter um repositório único de logs, força os MSs a usarem o mesmo formato de log, simplifica o monitoramento e o processo de análise.

Distributed tracing

Contexto: Em um sistema de MSs, é necessário gerar e coletar dados de log sobre interações entre os serviços.

Problema: Como gerar e coletar dados sobre interações entre os MS?

Solução: Gravar em log informações sobre as chamadas que serviços fazem entre si e entre clientes, bem como a quantidades de requisições feitas entre eles.

Vantagens: A coleta dos dados de requisições dos MSs permite determinar quem iniciou uma sequência de chamadas a um MS e dessa forma poder analisar a raiz de um problema. Os dados podem ser combinados com outras formas de log para melhorar o

processo de análise.

External monitor

Contexto: A natureza distribuída dos MSs resulta em uma complexa rede de interação entre os serviços, abrindo brechas para ataques de segurança contra as aplicações.

Problema: Como detectar ataques de segurança contra MSs?

Solução: O monitoramento e detecção de ataques é feito por um servidor externo aos MSs.

Vantagens: Permite realizar um monitoramento mais completo, com avaliações e ações sendo tomadas como resultado do estado geral do sistema.

Health check endpoint

Contexto: MSs podem ser implantados em qualquer lugar e podem estar indisponíveis por uma quantidade de tempo específica ou devido a um determinado contexto.

Problema: Como verificar se um MS está sendo executado e respondendo a requisições?

Solução: Adicionar pontos de checagem de MSs, responsáveis por verificar periodicamente o status e a habilidade dos serviços em responder requisições.

Vantagens: MSs que estão indisponíveis podem ser detectados e os clientes podem ser avisados para que não enviem requisições a estes serviços.

Desvantagens: A quantidade de comunicação entre os serviços aumenta devido a constante checagem de status, podendo afetar a performance. A quantidade de código fonte também aumenta, gerando um esforço maior de manutenção dos serviços.

Instrumentation

Contexto: Em um sistema de MSs, é necessário gerar e coletar dados de log em nível de host, plataforma e serviço.

Problema: Como gerar e coletar dados estáticos e de execução sobre cada MS?

Solução: Utilizar a instrumentação para coletar dados. A instrumentação pode ser de host, de plataforma e de serviço. Em todos os casos, é necessário a instalação de um agente que fica responsável por coletar informações.

Vantagens: Possibilita a coleta de dados em vários níveis no domínio de serviços (host, plataforma e serviço), permitindo que sejam armazenados posteriormente.

Local monitor

Contexto: A natureza distribuída dos MSs resulta em uma complexa rede de interação entre os serviços, abrindo brechas para ataques de segurança contra as aplicações.

Problema: Como detectar ataques de segurança contra MSs?

Solução: Executar um segundo serviço ao lado de cada MS, responsável por monitorar e detectar comportamentos anormais ou ataques em diferentes níveis do serviço.

Vantagens: Permite monitorar eventos de rede e evitar ataques contra a segurança dos MSs.

Logging

Contexto: Em um sistema de MSs, é necessário gerar e coletar dados de log sobre requisições e respostas enviadas aos clientes dos serviços.

Problema: Como gerar e coletar dados sobre requisições de entrada e saída dos MSs?

Solução: Gravar em log informações sensíveis sobre as requisições que entram e saem dos MSs. Cada entrada no log deve representar uma requisição específica, contendo data e hora da ocorrência, tempo de resposta, código de resposta, ID da instância do MS, URL do MS e método chamado. Estes dados são periodicamente agregados por ferramentas de log.

Vantagens: A coleta dos dados de requisições dos MSs permite analisar as informações que estão trafegando, o tempo de resposta, além de rastrear qual serviço foi responsável por determinado processamento.

Log aggregator

Contexto: MSs em execução geram informações sobre erros ocorridos, avisos, informações e outros dados, os quais são gravados em log em um formato padronizado.

Problema: Como entender o comportamento de uma aplicação e resolver problemas?

Solução: Utilizar um serviço de log centralizado que agrupa dados de log de cada instância de MSs.

Vantagens: Permite que usuários façam buscas nos dados de log e realizem análises, sendo possível configurar alertas quando determinadas mensagens aparecem.

None-aggregation processing of monitored data

Contexto: Diferentes tipos de dados sobre os MSs foram coletados através do monitoramento dos serviços.

Problema: Como processar e analisar os dados de monitoramento coletados?

Solução: Armazenar os dados coletados como foram coletados, ou seja, de forma detalhada.

Vantagens: Tendo os dados de log no formato em que foram coletados, é possível fazer análises mais detalhadas.

Proxy Circuit Breaker

Contexto: Em caso de falha, MSs devem ficar indisponíveis para novas requisições.

Problema: Como evitar que MSs continuem recebendo requisições em caso de falha?

Solução: Implementar um proxy para interceptar requisições entre serviços e evitar que um serviço receba requisições caso esteja com problemas. No caso do padrão proxy, este serviço é executado como um serviço separado, que fica entre o cliente e o MS, interceptando todas as requisições e bloqueando as que são enviadas para serviços indisponíveis.

Vantagens: MSs podem trabalhar em conjunto, causando dependências entre eles. Este padrão minimiza efeitos em cascata causados pela falha de um dos serviços na cadeia de dependência, auxiliando na disponibilidade dos MSs.

Service-side Circuit Breaker

Contexto: Em caso de falha, MSs devem ficar indisponíveis para novas requisições.

Problema: Como evitar que MSs continuem recebendo requisições em caso de falha?

Solução: Implementar um proxy para interceptar requisições entre serviços e evitar que um serviço receba requisições caso esteja com problemas. No caso do padrão service-side, este serviço roda no mesmo servidor que o MSs, interceptando todas as requisições que chegam ao serviço e bloqueando-as caso o MS esteja indisponível.

Vantagens: MSs podem trabalhar em conjunto, causando dependências entre eles. Este padrão minimiza efeitos em cascata causados pela falha de um dos serviços na cadeia de dependência, auxiliando na disponibilidade dos MSs.

Symmetric monitor

Contexto: Em um ambiente de MSs, é necessário monitorar os serviços para detectar ou antecipar falhas, objetivando tomar ações para que o sistema se recupere.

Problema: Como fazer o monitoramento dos serviços?

Solução: Fazer com que cada serviço seja monitorado por outros serviços vizinhos a ele.

Vantagens: Através da checagem de falhas, é possível evitar que requisições sejam encaminhadas para o serviço até que ele se recupere da falha.

Tabela 27 – Padrões arquiteturais para monitoramento e controle de falhas e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Aggregation processing of monitored data	monitored data processing	monitorability	++	[P4]

Continua na próxima página

Tabela 27 – continuado da página anterior

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Application metrics	less development complexity	agility	-	[P11]
Application metrics	metrics measurement	performance	++	[P11]
Arbital monitor	fail isolation	availability	++	[P4]
Bulkhead	fail isolation	availability	+	[P3, P13]
Centralized monitor	fail isolation	availability	++	[P4]
Centralized storage of monitored data	monitored data storing	monitorability	++	[P4]
Client-side circuit breaker	fail isolation	availability	++	[P4]
Correlation ID	traceability	monitorability	++	[G5]
Decentralized storage of monitored data	monitored data storing	monitorability	++	[P4]
Distributed logging	monitored data storing	monitorability	++	[P11]
Distributed tracing	communication monitoring	monitorability	++	[P4]
External monitor	security monitoring	monitorability	++	[P4]
External monitor	attack detection	security	++	[P4]
Health check endpoint	fail isolation	availability	++	[P6, P11]
Health check endpoint	efficient health check	performance	-	[P6, P11]
Instrumentation	communication monitoring	monitorability	++	[P4]
Instrumentation	resources use monitoring	monitorability	++	[P4]
Local monitor	security monitoring	monitorability	++	[P4]
Local monitor	attack detection	security	++	[P4]
Log aggregator	monitored data storing	monitorability	++	[P6, P8], [G19]
Logging	communication monitoring	monitorability	++	[P4]
None-aggregation processing of monitored data	monitored data processing	monitorability	++	[P4]
Proxy circuit breaker	fail isolation	availability	++	[P4]
Service-side circuit breaker	fail isolation	availability	++	[P4]
Symmetric monitor	fail isolation	availability	++	[P4]

Fonte: Produzido pelo autor

1.2.15 Padrões para Registro de Serviços

Nesta categoria estão os padrões que tratam sobre as formas de implementar o registro dos serviços nos servidores de registro, para que possam ser descobertos pelos clientes.

Manual registry

Contexto: MSs podem ter várias instâncias implantadas e a quantidade dessas instâncias pode mudar com o passar do tempo. A localização dos serviços em execução deve ser armazenada em um serviço central para que outros MSs possam consultar.

Problema: Como implementar o registro dos serviços em execução em um serviço central?

Solução: Implementar o registro dos serviços de forma manual, onde um usuário inclui as informações no serviço central.

Vantagens: Auxilia na disponibilidade dos serviços, pois, caso o usuário detecte uma falha em um MS, pode removê-lo do serviço de registro para que os clientes não o utilizem. O registro manual facilita que MSs de outros sistemas sejam incluídos no serviço de registros conforme a necessidade dos usuários.

Desvantagens: Como o registro é feito manualmente, pode demorar para que falhas sejam detectadas.

Self-registry

Contexto: MSs podem ter várias instâncias implantadas e a quantidade dessas instâncias pode mudar com o passar do tempo. A localização dos serviços em execução deve ser armazenada em um serviço central para que outros MSs possam consultar.

Problema: Como implementar o registro dos serviços em execução em um serviço central?

Solução: Cada MS pode se registrar ou se remover do registro no momento em que é inicializado ou desligado.

Vantagens: Devido ao registro automático e monitoramento dos serviços, a detecção de falhas ocorre com rapidez, evitando que MSs inválidos sejam acessados pelos clientes.

Third-party registry

Contexto: MSs podem ter várias instâncias implantadas e a quantidade dessas instâncias pode mudar com o passar do tempo. A localização dos serviços em execução deve ser armazenada em um serviço central para que outros MSs possam consultar.

Problema: Como implementar o registro dos serviços em execução em um serviço central?

Solução: Cada MS pode se registrar ou se remover do registro no momento em que é inicializado ou desligado. Essa tarefa é implementada através de componentes de terceiros, os quais são implantados na arquitetura de MSs.

Vantagens: Devido ao registro automático e monitoramento dos serviços, a detecção de falhas ocorre com rapidez, evitando que MSs inválidos sejam acessados pelos clientes.

Tabela 28 – Padrões arquiteturais para registro de serviços e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Manual registry	service registry	availability	++	[P4]
Manual registry	service interoperability	reusability	+	[P4]
Self-registry	service registry	availability	++	[P4]

Continua na próxima página

Tabela 28 – continuado da página anterior

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Third-party registry	service registry	availability	++	[P4]

Fonte: Produzido pelo autor

1.2.16 Padrões para Reúso de MS

Nesta categoria estão os padrões que podem ser aplicados visando a padronização e diminuição de retrabalho ao implementar novos serviços.

Microservice chassis

Contexto: No desenvolvimento de um sistema em MSs, comumente é necessário gastar um tempo significativo escrevendo lógica para construção e testes dos serviços. Além disso, é necessário desenvolver soluções para segurança, configuração externalizada, log e checagem de falhas. Essas tarefas muitas vezes se repetem para cada serviço.

Problema: Como diminuir a necessidade de execução de tarefas repetidas no desenvolvimento de um sistema em MSs?

Solução: Criar um framework que pode servir como base para o desenvolvimento dos MSs. Este framework deve conter todas as implementações que são comuns a todos os MSs do sistema.

Vantagens: Permite uma maior agilidade no desenvolvimento dos serviços. *Desvantagens:* Deve haver um framework para cada linguagem utilizada na programação dos MSs, o que pode ser um obstáculo na adoção de novas tecnologias.

Tabela 29 – Padrões arquiteturais para reúso e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Chassis	use of service templates	agility	++	[G21]
Chassis	service templates	maintainability	++	[G16]
Chassis	service templates	reusability	++	[G16]

Fonte: Produzido pelo autor

1.2.17 Padrões para Segurança

Nesta categoria estão os padrões que se aplicam aos aspectos de segurança dos serviços.

Client certificate

Contexto: Em um sistema é necessário proteger dados e informações para que somente entidades autorizadas tenham acesso. A natureza distribuída dos MSs faz com que a interação entre serviços se torne bastante complexa. Pessoas mal-intencionadas podem se aproveitar desta complexidade para realizar ataques contra as aplicações na tentativa de obter acesso.

Problema: Como proteger os canais de comunicação entre os serviços?

Solução: Utilizar certificados digitais para autenticar os hosts participantes de uma troca de mensagens e criptografar as mensagens trocadas, protegendo, dessa forma, o canal de comunicação entre eles.

Vantagens: A utilização de certificados digitais traz segurança na troca de mensagens entre os MSs, evitando que entidades externas tenham acesso ao seu conteúdo.

Federated identity

Contexto: Em um sistema é necessário proteger dados e informações para que somente entidades autorizadas tenham acesso. A natureza distribuída dos MSs faz com que a interação entre serviços se torne bastante complexa. Pessoas mal-intencionadas podem se aproveitar desta complexidade para realizar ataques contra as aplicações na tentativa de obter acesso.

Problema: Como proteger os canais de comunicação entre os serviços e permitir que somente pessoas autorizadas tenham acesso aos serviços?

Solução: Delegar a autenticação dos usuários para serviços de terceiros. Um serviço pode requerer informações de autenticação a uma entidade federada externa, a qual retorna um token com dados sobre o usuário. Este token pode ser utilizado em conjunto com certificados digitais para criptografia do canal de comunicação e autenticação dos usuários envolvidos na comunicação. O serviço que recebe o token pode consultar a entidade federada externa para validação dos dados e autenticação.

Vantagens: A utilização de tokens em conjunto com certificados digitais traz segurança na troca de mensagens entre os MSs, evitando que entidades externas tenham acesso ao seu conteúdo. As entidades federadas externas facilitam a implementação da segurança, evitando a necessidade de gerenciamento de uma entidade federada local.

Gatekeeper

Contexto: Aplicações expõe suas funcionalidades para os clientes através da aceitação e processamento de requisições e normalmente incluem em seu código fonte funções para fazer autenticação e validação, processamento de requisições, acesso a armazenamento e outros serviços.

Problema: Um usuário malicioso pode comprometer o sistema e ganhar acesso ao ambiente que hospeda a aplicação, aos mecanismos de segurança, serviços e dados.

Solução: Para minimizar o risco de usuários ganharem acesso a informações sensíveis, deve-se desacoplar hosts ou tarefas que expõe interfaces públicas do código que processa requisições e acessa áreas de armazenamento. Isso pode ser conseguido através do uso de façades ou tarefas dedicadas que interagem com o cliente e manipulam as requisições.

Vantagens: Permite minimizar o risco de ataques contra as aplicações, atuando como um firewall de uma rede, examinando requisições e fazendo decisões sobre permitir ou não requisições sejam aceitas e processadas.

Key exchanged-based communication authentication

Contexto: Em um sistema é necessário proteger dados e informações para que somente entidades autorizadas tenham acesso. A natureza distribuída dos MSs faz com que a interação entre serviços se torne bastante complexa. Pessoas mal-intencionadas podem se aproveitar desta complexidade para realizar ataques contra as aplicações na tentativa de obter acesso.

Problema: Como proteger os canais de comunicação entre os serviços?

Solução: Utilizar a troca de chaves para autenticar os hosts participantes de um processo de uma comunicação entre dois serviços, utilizando-as para criptografar as mensagens. Apenas os participantes da comunicação conhecem a chave e podem descriptografar as mensagens trocadas.

Vantagens: A utilização de chaves de criptografia traz segurança na troca de mensagens entre os MSs, evitando que entidades externas tenham acesso ao seu conteúdo.

Tabela 30 – Padrões arquiteturais para segurança e RNFs impactados

Padrão	RNF impactado	RNF Pai	Impacto	Referências
Client certificate	communication security	security	+	[P4]
Federated identity	easier security management	manageability	++	[G10]
Federated identity	communication security	security	+	[P4]
Gatekeeper	attack detection	security	++	[P12, P13], [G11]
Key exchanged-based communication authentication	communication security	security	+	[P4]

Fonte: Produzido pelo autor

1.3 Grafos de Interdependência de softgoals - SIGs

Nesta seção são apresentados os grafos de interdependência de *softgoal*, construídos a partir dos RNFs, padrões arquiteturais (operacionalizações) e seus impactos nos requisitos. Devido ao grande número de padrões, foram construídos SIGs individuais para cada RNF

pai, de forma a diminuir o tamanho do grafo e possibilitar uma visualização comprehensível ao leitor.

Para auxiliar na compreensão dos grafos, a seguir é apresentado um resumo dos tipos de contribuições possíveis no NFR Framework.

AND: todos os *softgoals* descendentes devem ser satisfeitos para que o *softgoal* pai seja satisfeito.

OR: se qualquer *softgoal* descendente for satisfeito, o *softgoal* pai também será.

MAKE (++): se um *softgoal* descendente for satisfeito, o *softgoal* pai também o será de forma suficientemente positiva.

BREAK (-) se um *softgoal* descendente for satisfeito, o *softgoal* pai será completamente negado, ou seja, a satisfação do descendente faz com que seu pai não seja alcançado.

HELP (+): se um *softgoal* descendente for parcialmente satisfeito, o *softgoal* pai também o será, de forma parcial.

HURT (-): se um *softgoal* descendente for satisfeito, o *softgoal* pai será parcialmente negado, ou seja, a satisfação do descendente prejudica o alcance de seu pai.

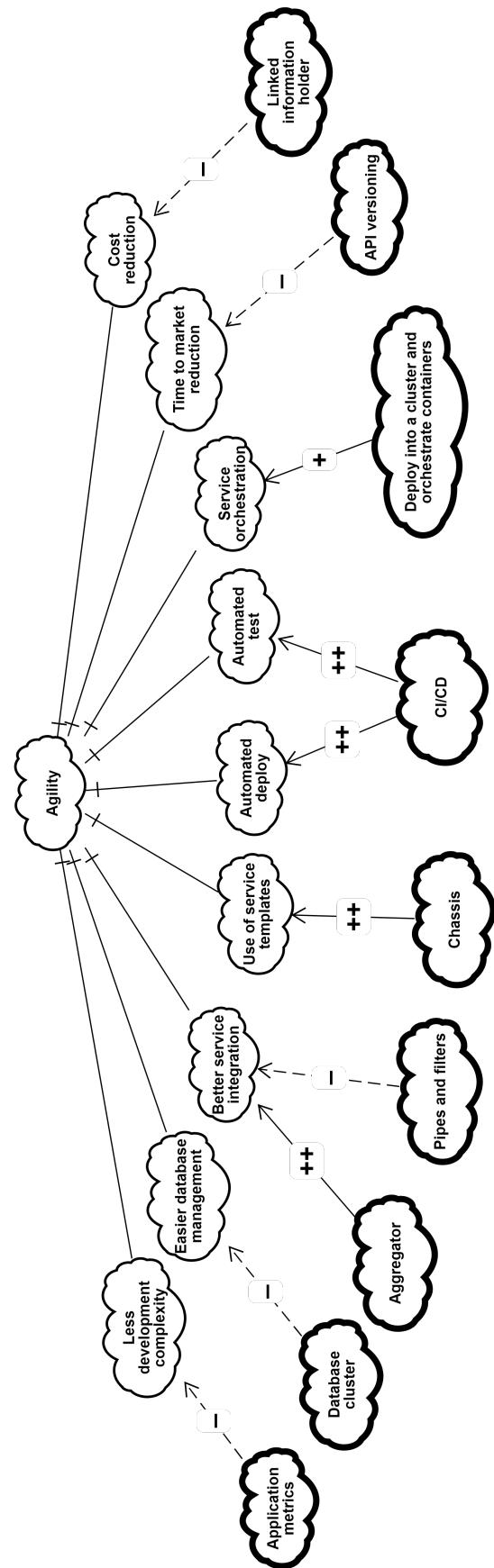
UNKNOWN (?): a contribuição em um *softgoal* ascendente e seu pai é desconhecida, podendo ser tanto positiva quanto negativa.

EQUALS: para que um *softgoal* descendente seja satisfeito, seu pai também deverá ser, e, para que um *softgoal* descendente seja negado, seu pai deverá ser negado.

SOME+: quando se sabe que um *softgoal* ascendente contribui de forma positiva com seu pai, mas não se sabe a extensão dessa contribuição, pode-se utilizar esta notação.

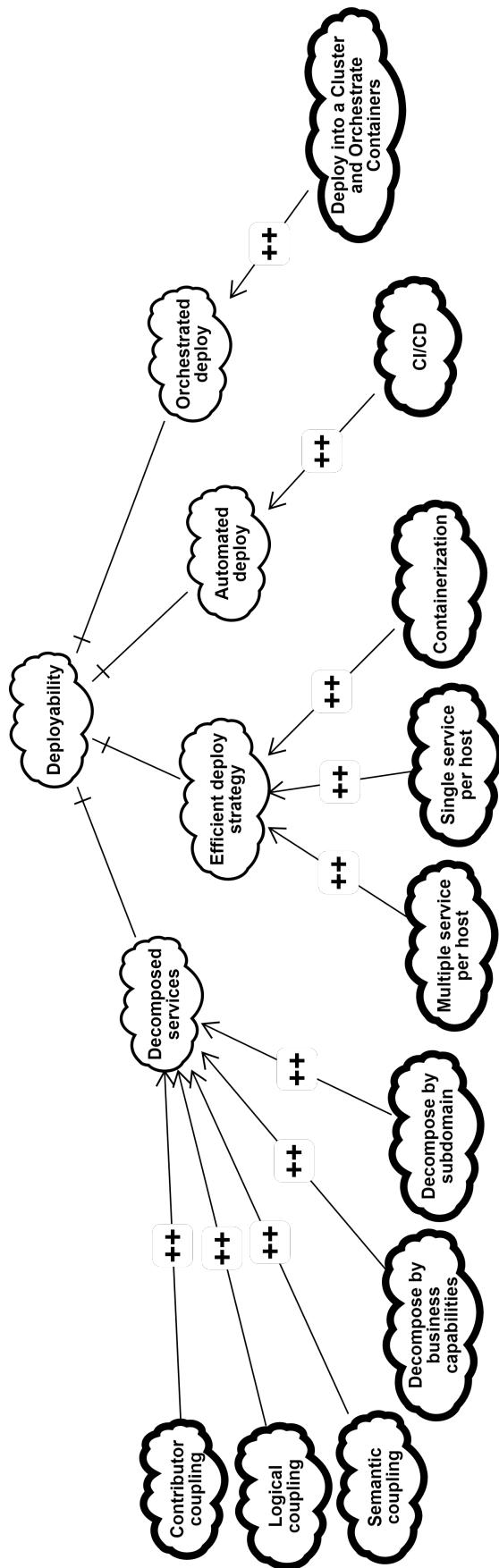
SOME-: quando se sabe que um *softgoal* ascendente contribui de forma negativa com seu pai, mas não se sabe a extensão dessa contribuição, pode-se utilizar esta notação.

Figura 1 – SIG para o RNF Agility



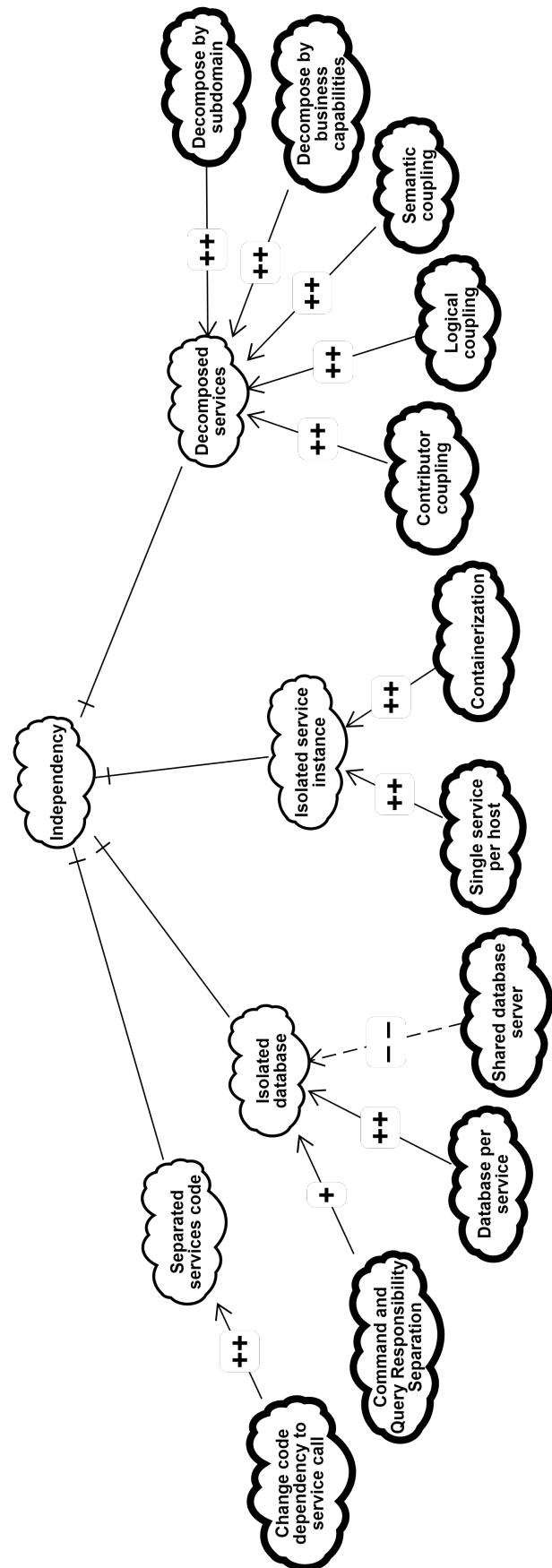
Fonte: Produzido pelo autor

Figura 3 – SIG para o RNF Deployability



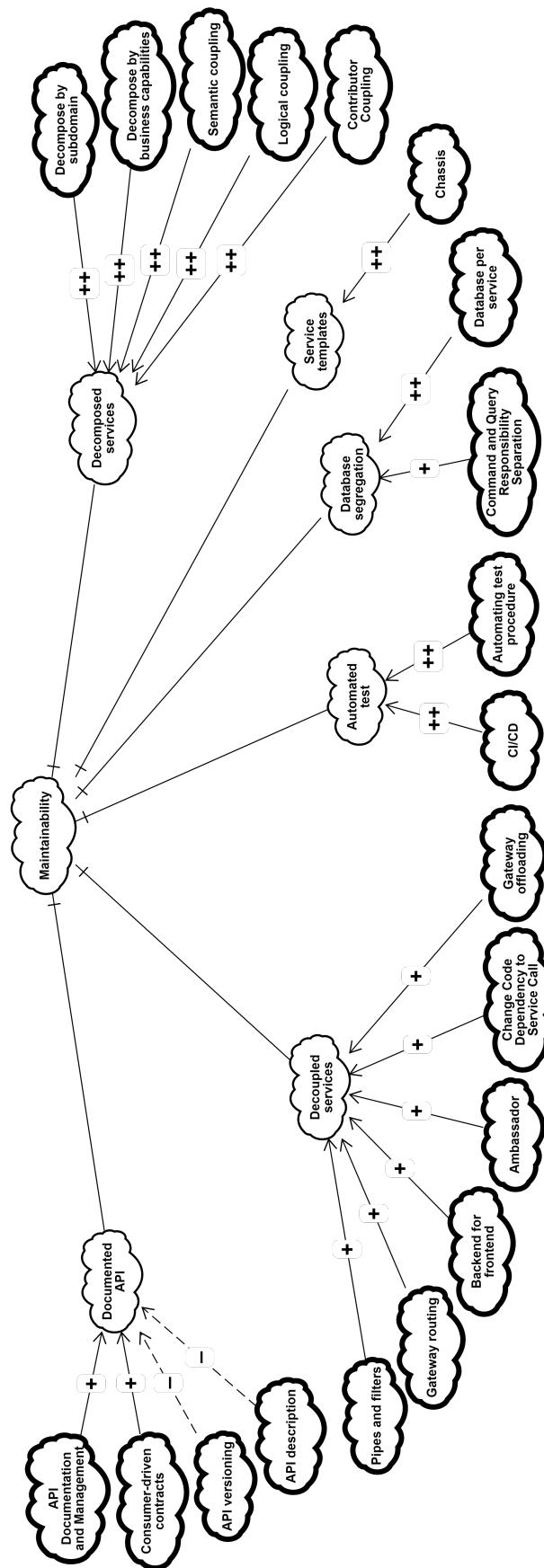
Fonte: Produzido pelo autor

Figura 4 – SIG para o RNF Independency



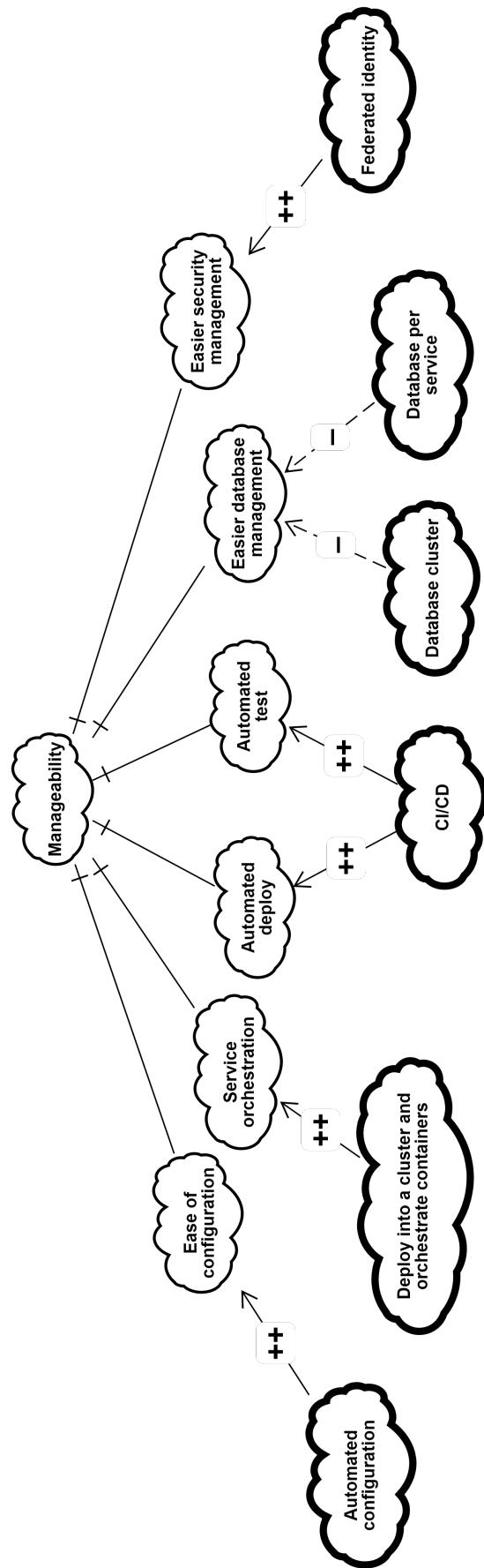
Fonte: Produzido pelo autor

Figura 5 – SIG para o RNF Maintainability



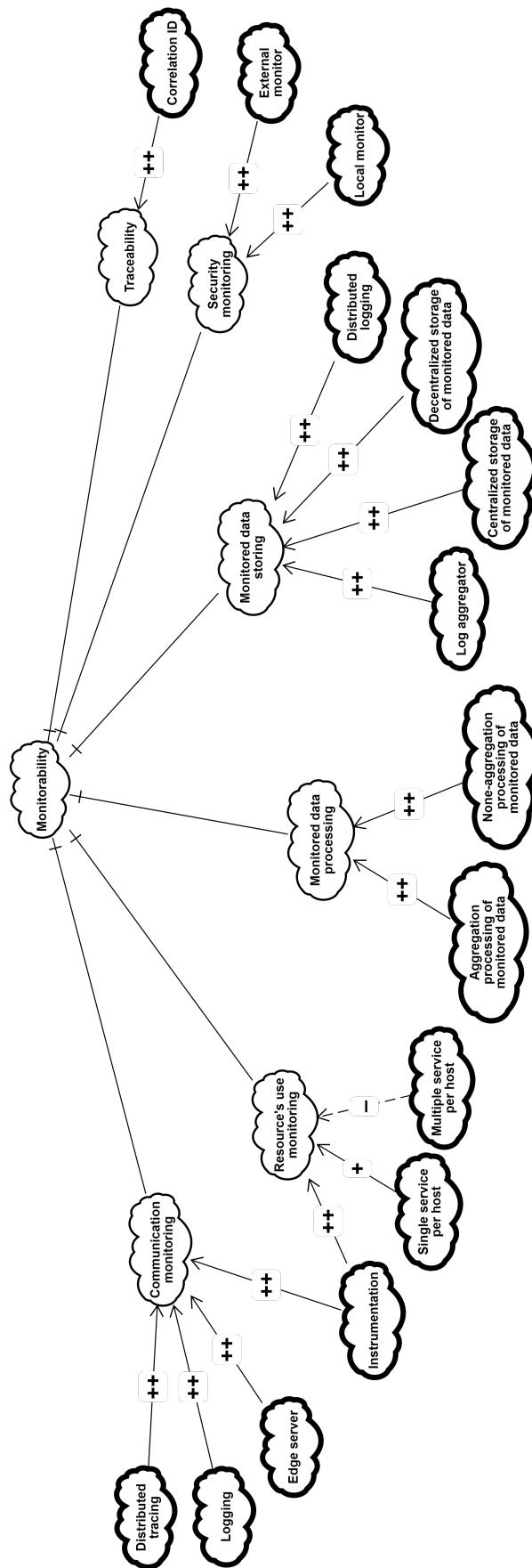
Fonte: Produzido pelo autor

Figura 6 – SIG para o RNF Manageability



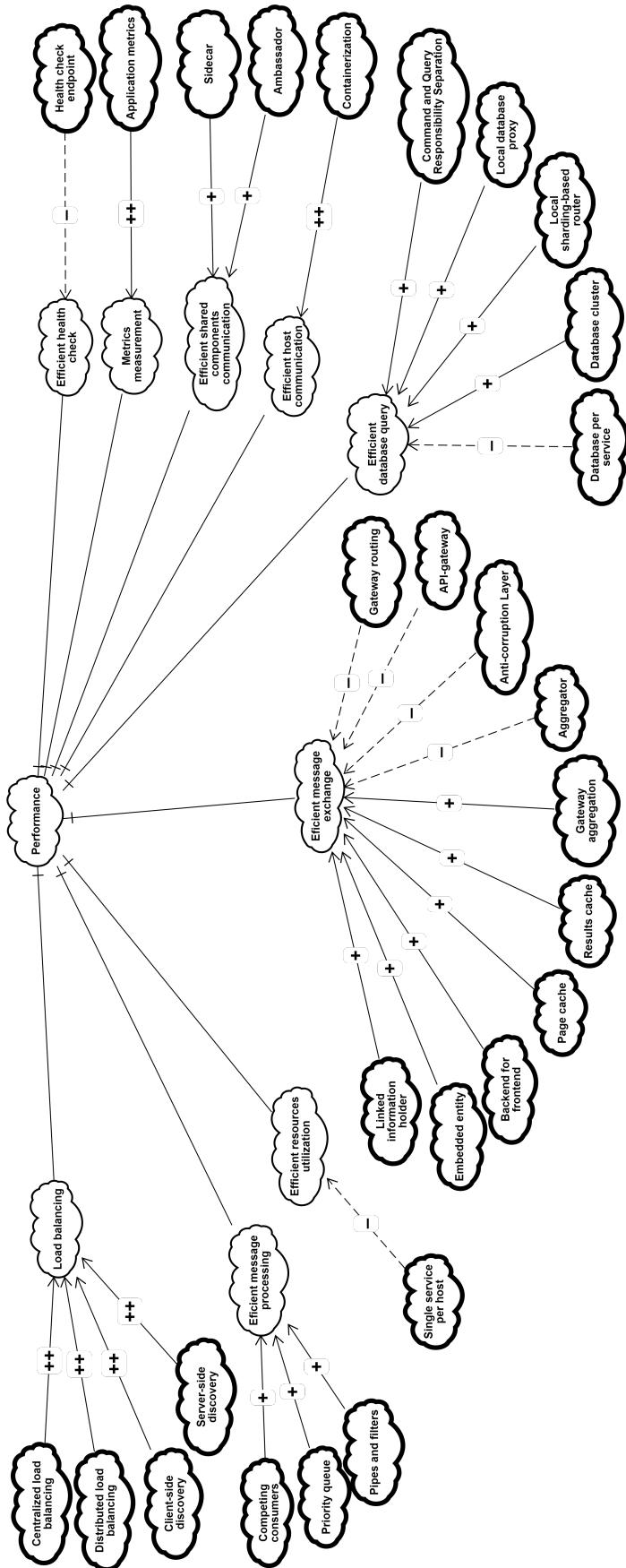
Fonte: Produzido pelo autor

Figura 7 – SIG para o RNF Monitorability



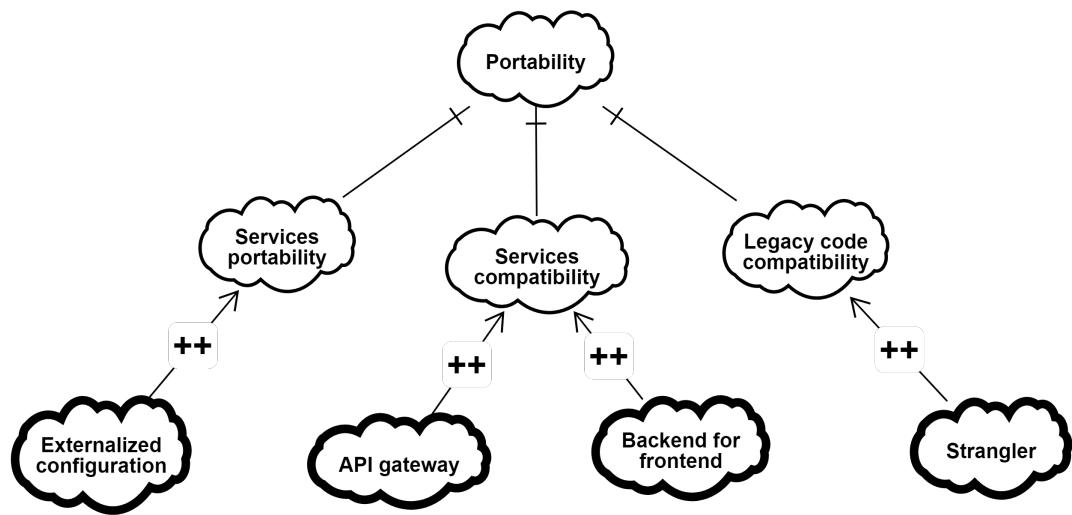
Fonte: Produzido pelo autor

Figura 8 – SIG para o RNF Performance



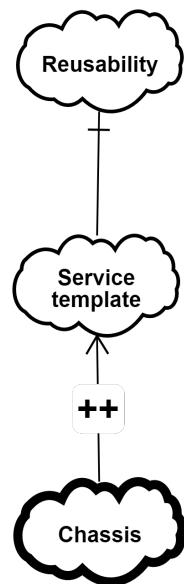
Fonte: Produzido pelo autor

Figura 9 – SIG para o RNF Portability



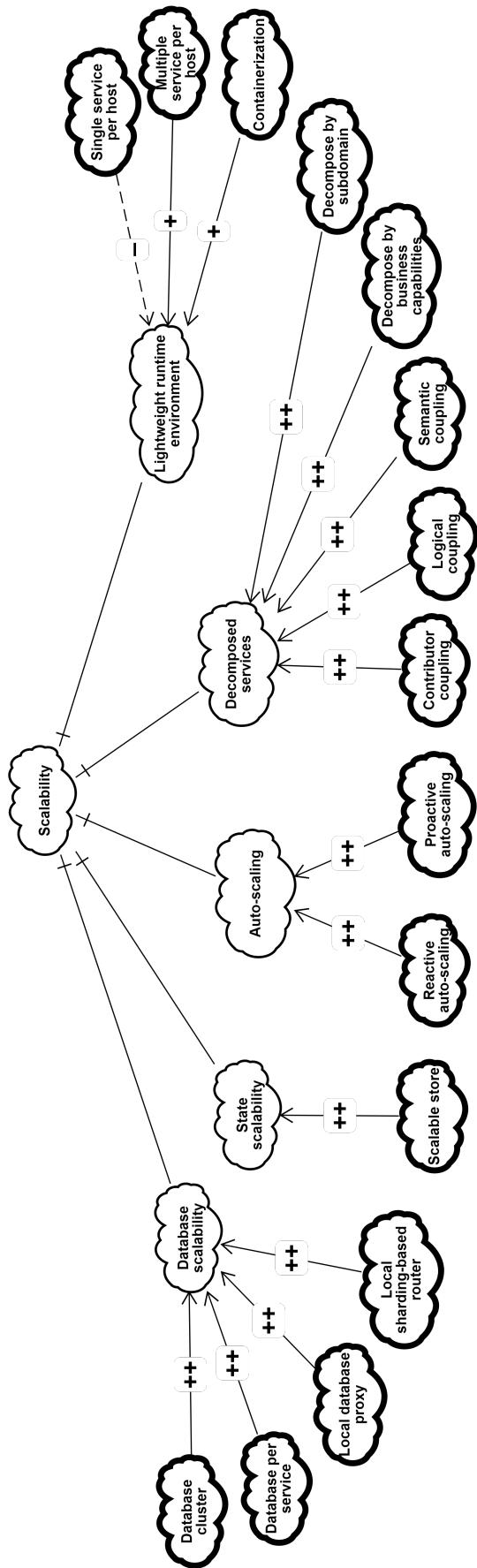
Fonte: Produzido pelo autor

Figura 10 – SIG para o RNF Reusability



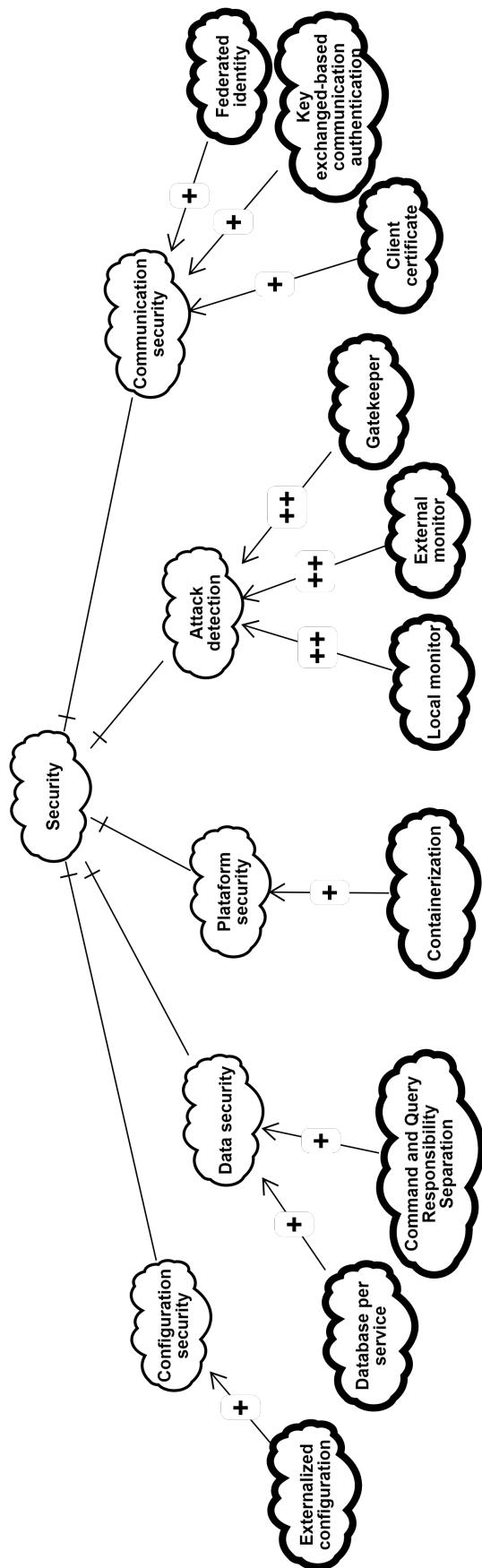
Fonte: Produzido pelo autor

Figura 11 – SIG para o RNF Scalability



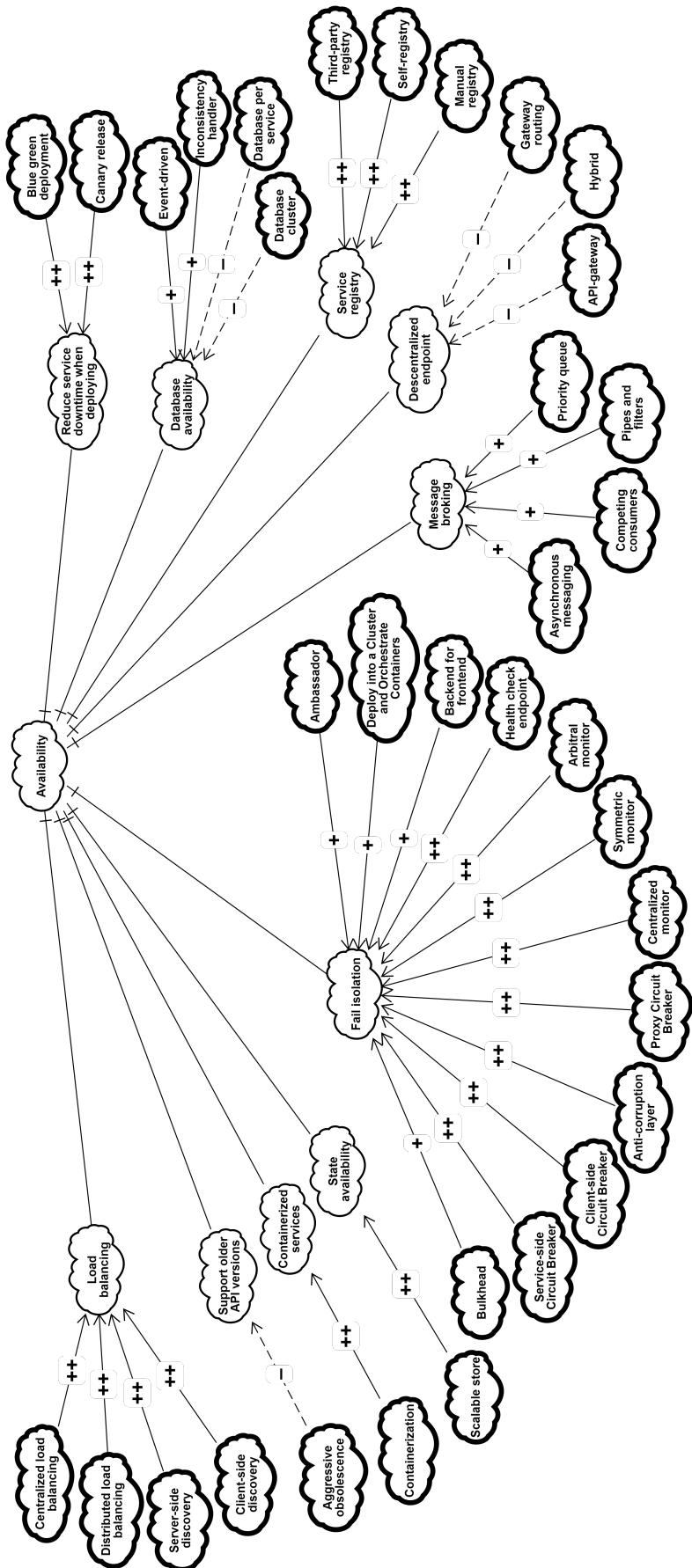
Fonte: Produzido pelo autor

Figura 12 – SIG para o RNF Security



Fonte: Produzido pelo autor

Figura 2 – SIG para o RNF Availability



Fonte: Produzido pelo autor

Referências

- [Bass, Clements e Kazman 2003] BASS, L.; CLEMENTS, P.; KAZMAN, R. Software Architecture in Practice , Second Edition. *Software Architecture*, p. 560, 2003. Citado 7 vezes nas páginas [8](#), [9](#), [10](#), [11](#), [12](#), [13](#) e [14](#).
- [Bass, Weber e Liming 2015] BASS, L.; WEBER, I.; LIMING, Z. *DevOps: A Software Architect's Perspective*. [S.l.]: Addison-Wesley, 2015. Citado na página [9](#).
- [Bogner et al. 2019] BOGNER, J. et al. Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges. *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, IEEE, p. 546–556, 2019. Citado na página [9](#).
- [Cojocaru, Oprescu e Uta 2019] COJOCARU, M. D.; OPRESCU, A.; UTA, A. Attributes assessing the quality of microservices automatically decomposed from monolithic applications. *Proceedings - 2019 18th International Symposium on Parallel and Distributed Computing, ISPDC 2019*, n. 1, p. 84–93, 2019. Citado na página [10](#).
- [Da Silva et al. 2019] Da Silva, M. A. P. et al. A microservice-based approach for increasing software reusability in health applications. *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, v. 2019-Novem, 2019. ISSN 21615330. Citado na página [13](#).
- [Li et al. 2021] LI, S. et al. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology*, Elsevier B.V., v. 131, p. 106449, 2021. ISSN 09505849. Disponível em: <<https://doi.org/10.1016/j.infsof.2020.106449>>. Citado 4 vezes nas páginas [8](#), [11](#), [12](#) e [14](#).
- [Mairiza, Zowghi e Nurmuliani 2010] MAIRIZA, D.; ZOWGHI, D.; NURMULIANI, N. An investigation into the notion of non-functional requirements. *Proceedings of the ACM Symposium on Applied Computing*, n. May, p. 311–317, 2010. Citado na página [10](#).
- [Mendonça et al. 2018] MENDONÇA, N. C. et al. Generality vs. reusability in architecture-based self-adaptation: The case for self-adaptive microservices. *ACM International Conference Proceeding Series*, 2018. Citado na página [13](#).
- [Mišić, Ramač e Mandić 2017] MIŠIĆ, B.; RAMAČ, R.; MANDIĆ, V. Do the microservices improve the agility of software development teams ? *XVII International Scientific Conference on Industrial Systems*, n. October 2017, p. 170–175, 2017. Citado na página [7](#).

[O'Brien, Merson e Bass 2007] O'BRIEN, L.; MERSON, P.; BASS, L. Quality attributes for service-oriented architectures. *Proceedings - ICSE 2007 Workshops: International Workshop on Systems Development in SOA Environments, SDSOA '07*, p. 3–9, 2007. Citado 3 vezes nas páginas [8](#), [12](#) e [13](#).

[Ruben Prieto-Diaz 1993] Ruben Prieto-Diaz. Status Report: Software Reusability. *IEEE Software*, v. 10, n. 3, p. 61–66, 1993. ISSN 07407459. Citado na página [13](#).

[Taibi e Lenarduzzi 2018] TAIBI, D.; LENARDUZZI, V. On the Definition of Microservice Bad Smells. *IEEE Software*, v. 35, n. 3, p. 56–62, 2018. ISSN 07407459. Citado na página [10](#).

[Toffetti et al. 2015] TOFFETTI, G. et al. An architecture for self-managing microservices. *Proceedings: AIMC 2015 - Automated Incident Management in Cloud, 1st International Workshop, in conjunction with EuroSYS 2015*, p. 19–24, 2015. Citado na página [11](#).

[Vural e Koyuncu 2021] VURAL, H.; KOYUNCU, M. Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice? *IEEE Access*, p. 32721–32733, 2021. ISSN 21693536. Citado na página [10](#).

Fontes Consideradas na Identificação dos Padrões Arquiteturais

- [P1] J Bogner, S Wagner, and A Zimmermann. Using architectural modifiability tactics to examine evolution qualities of Service- and Microservice-Based Systems: An approach based on principles and patterns. *Software-Intensive Cyber-Physical Systems*, 34(2-3):141–149, 2019. Citado na página [17](#).
- [P2] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges. *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pages 546–556, 2019. Citado 3 vezes nas páginas [17](#), [22](#) e [44](#).
- [P3] Vaidas Giedrimas, Samir Omanovic, and Dino Alic. *The aspect of resilience in microservices-based software design*, volume 11176 LNCS. Springer International Publishing, 2018. Citado 3 vezes nas páginas [17](#), [43](#) e [51](#).
- [P4] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology*, 131:106449, 2021. Citado 13 vezes nas páginas [22](#), [23](#), [24](#), [25](#), [28](#), [33](#), [36](#), [38](#), [50](#), [51](#), [52](#), [53](#) e [55](#).
- [P5] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns - Balancing compatibility and extensibility across service life cycles. *ACM International Conference Proceeding Series*, 2019. Citado 2 vezes nas páginas [21](#) e [22](#).
- [P6] Gaston Marquez and Hernan Astudillo. Actual Use of Architectural Patterns in Microservices-Based Open Source Projects. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2018-Decem:31–40, 2018. Citado 6 vezes nas páginas [23](#), [24](#), [25](#), [28](#), [38](#) e [51](#).
- [P7] Felipe Osses, Gastón Márquez, and Hernán Astudillo. An exploratory study of academic architectural tactics and patterns in microservices: A systematic literature review. *Avances en Ingeniería de Software a Nivel Iberoamericano, CIBSE 2018*, (February):71–84, 2018. Citado 3 vezes nas páginas [17](#), [38](#) e [43](#).

- [P8]Tiago Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A Survey on the Adoption of Patterns for Engineering Software for the Cloud. *IEEE Transactions on Software Engineering*, 5589(c):1–13, 2021. Citado 2 vezes nas páginas [38](#) e [51](#).
- [P9]Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018-Janua(Closer 2018):221–232, 2018. Citado 7 vezes nas páginas [17](#), [26](#), [28](#), [34](#), [35](#), [38](#) e [43](#).
- [P10]Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. *Continous Architecting with Microservices and DevOps* :, volume 1. Springer International Publishing, 2019. Citado 4 vezes nas páginas [17](#), [28](#), [38](#) e [43](#).
- [P11]Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaïdi, Jean Privat, and Yann Gaël Guéhéneuc. On the Study of Microservices Antipatterns: A Catalog Proposal. *ACM International Conference Proceeding Series*, (July), 2020. Citado 7 vezes nas páginas [18](#), [22](#), [23](#), [26](#), [28](#), [38](#) e [51](#).
- [P12]J A Valdivia, X Limon, and K Cortes-Verdin. Quality attributes in patterns related to microservice architecture: A Systematic Literature Review. In *Proceedings - 2019 7th International Conference in Software Engineering Research and Innovation, CONISOFT 2019*, pages 181–190, 2019. Citado 5 vezes nas páginas [17](#), [31](#), [38](#), [43](#) e [55](#).
- [P13]J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, and J. O. Ocharán-Hernández. Patterns Related to Microservice Architecture: a Multivocal Literature Review. *Programming and Computer Software*, 46(8):594–608, 2020. Citado 10 vezes nas páginas [17](#), [23](#), [24](#), [25](#), [28](#), [31](#), [38](#), [43](#), [51](#) e [55](#).
- [P14]Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. Introduction to microservice API patterns (MAP). *OpenAccess Series in Informatics*, 78(4):1–4, 2020. Citado na página [22](#).

Fontes Consideradas na Complementação das Informações sobre os Padrões Arquiteturais

- [G1]Marty Abbott. Microservice aggregator pattern. Disponível em: <https://akfpartners.com/growth-blog/microservice-aggregator-pattern>. Acesso em: 05 de outubro de 2021. Citado na página 17.
- [G2]Arcitura. Container sidecar. Disponível em: https://patterns.arcitura.com/microservice-patterns/design_patterns/container_sidecar. Acesso em: 29 de outubro de 2021. Citado na página 43.
- [G3]Arcitura. Microservice ambassador. Disponível em: https://patterns.arcitura.com/microservice-patterns/design_patterns/microservice_ambassador. Acesso em: 09 de outubro de 2021. Citado na página 43.
- [G4]Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. Microservices migration patterns. *Software - Practice and Experience*, 48(11):2019–2042, 2018. Citado na página 43.
- [G5]Kyle Brown and Bobby Woolf. *Implementation patterns for microservices architectures*. 2016. Citado na página 51.
- [G6]Cloud Fondry. Using blue-green deployment to reduce downtime and risk. Disponível em: <https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html>. Acesso em: 22 de outubro de 2021. Citado na página 38.
- [G7]Foutse Khomh and S. Amirhossein Abtahizadeh. Understanding the impact of cloud patterns on performance and energy consumption. *Journal of Systems and Software*, 141:151–170, 2018. Citado na página 28.
- [G8]Microsoft. Anti-corruption layer pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer>. Acesso em: 05 de outubro de 2021. Citado na página 17.
- [G9]Microsoft. Competing consumers pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/competing-consumers>. Acesso em: 10 de outubro de 2021. Citado na página 43.
- [G10]Microsoft. Federated identity pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/federated-identity>. Acesso em: 19 de outubro de 2021. Citado na página 55.

[G11] Microsoft. Gatekeeper pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/gatekeeper>. Acesso em: 19 de outubro de 2021. Citado na página 55.

[G12] Microsoft. Gateway offloading pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading>. Acesso em: 29 de outubro de 2021. Citado na página 43.

[G13] Microsoft. Gateway routing pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-routing>. Acesso em: 29 de outubro de 2021. Citado na página 43.

[G14] Microsoft. Pipes and filters pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters>. Acesso em: 11 de outubro de 2021. Citado na página 43.

[G15] Microsoft. Priority queue pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/priority-queue>. Acesso em: 11 de outubro de 2021. Citado na página 43.

[G16] Akhil Raj. Microservices chassis pattern. Disponível em: <https://dzone.com/articles/ms-chassis-pattern>. Acesso em: 29 de outubro de 2021. Citado na página 53.

[G17] Chris Richardson. Pattern: Event-driven architecture. Disponível em: <https://microservices.io/patterns/data/event-driven-architecture.html>. Acesso em: 22 de outubro de 2021. Citado na página 28.

[G18] Chris Richardson. Pattern: Externalized configuration. Disponível em: <https://microservices.io/patterns/externalized-configuration.html>. Acesso em: 09 de outubro de 2021. Citado na página 31.

[G19] Chris Richardson. Pattern: Log aggregation. Disponível em: <https://microservices.io/patterns/observability/application-logging.html>. Acesso em: 18 de outubro de 2021. Citado na página 51.

[G20] Chris Richardson. Pattern: Messaging. Disponível em: <https://microservices.io/patterns/communication-style/messaging.html>. Acesso em: 09 de outubro de 2021. Citado na página 43.

[G21] Chris Richardson. Pattern: Microservice chassis. Disponível em: <https://microservices.io/patterns/microservice-chassis.html>. Acesso em: 29 de outubro de 2021. Citado na página 53.

[G22]Chris Richardson. Pattern: Multiple service instances per host. Disponível em: <https://microservices.io/patterns/deployment/multiple-services-per-host.html>. Acesso em: 18 de outubro de 2021. Citado na página 38.

[G23]Danilo Sato. Canary release. Disponível em: <https://martinfowler.com/bliki/CanaryRelease.html>. Acesso em: 23 de outubro de 2021. Citado na página 38.