Um Blueprint Simplificado para Sistemas de IA Resilientes: Integrando MCP com Arquitetura Hexagonal em um MVP de Projeto Único

Sumário Executivo

Este relatório refina o blueprint original "Arquitetura Hexagonal com MCP Detalhado" para um Produto Mínimo Viável (MVP).[1] Ele propõe uma simplificação estratégica de um projeto Maven multi-módulo complexo para uma estrutura de projeto único mais ágil. O principal desafio de uma estrutura de projeto único é o risco de decadência arquitetônica devido à perda da aplicação de dependências em tempo de compilação. Este blueprint atualizado aborda isso diretamente, introduzindo o

ArchUnit como um mecanismo crítico em tempo de teste para impor programaticamente os limites estritos entre as camadas de domínio, aplicação e adaptador.[2] O resultado é uma arquitetura de MVP robusta, de fácil manutenção e testável que preserva os princípios fundamentais da Arquitetura Hexagonal e a integração do Protocolo de Contexto do Modelo (MCP), ao mesmo tempo que acelera a velocidade de desenvolvimento inicial. Este documento fornece um caminho concreto e pronto para produção para implementação usando Java, Quarkus e as melhores práticas de segurança e tolerância a falhas.

Seção 1: Conceitos Fundamentais: Uma Visão Geral Simbiótica

Esta seção estabelece a base teórica para a arquitetura. É crucial reiterar esses conceitos porque, em uma estrutura de projeto único, um profundo entendimento dos princípios é a primeira linha de defesa contra violações arquitetônicas. A clareza sobre os limites e responsabilidades de cada padrão é o que permite que a simplificação do projeto não se transforme em um acoplamento indesejado.

1.1. O Protocolo de Contexto do Modelo (MCP) Decodificado

O Protocolo de Contexto do Modelo (MCP) é um padrão aberto projetado para resolver um desafio fundamental na era da IA generativa: como as aplicações podem fornecer contexto de forma padronizada para Modelos de Linguagem Grandes (LLMs).[5] A proliferação de modelos de IA e ferramentas externas cria um "problema

M×N", onde cada uma das M aplicações de IA precisaria de uma integração personalizada para cada uma das N capacidades externas. O MCP transforma isso em um problema "M+N" muito mais gerenciável, fornecendo uma interface padrão: cada aplicação de IA implementa o lado do cliente do MCP uma vez, e cada

ferramenta ou fonte de dados implementa o lado do servidor uma vez.[6] Essencialmente, o MCP atua como uma "porta USB-C para aplicações de IA", oferecendo uma maneira consistente e universal de conectar modelos a capacidades externas.[7]

Arquitetura Cliente-Servidor

No seu cerne, o MCP opera em uma arquitetura cliente-servidor, que desacopla a aplicação que consome o contexto da fonte que o fornece.[5] Os componentes principais são:

- **Host:** A aplicação de IA voltada para o usuário com a qual os usuários finais interagem diretamente. Exemplos incluem IDEs aprimorados por IA, assistentes de desktop ou aplicações personalizadas. O Host inicia as conexões com os Servidores MCP e orquestra o fluxo geral.[6]
- **Cliente (Client):** Um componente dentro do Host que gerencia a comunicação com um Servidor MCP específico. Cada Cliente mantém uma conexão 1:1 com um único Servidor, atuando como um intermediário entre a lógica do Host e o Servidor externo.[8]
- **Servidor (Server):** Um programa ou serviço externo que expõe capacidades (Ferramentas, Recursos, Prompts) através do protocolo MCP. Estes são os provedores de contexto.[8]

Capacidades Essenciais (A Camada de Contexto)

Os Servidores MCP fornecem contexto através de três mecanismos primários, permitindo uma rica interação com o LLM [7]:

1. **Recursos (Resources):** Fontes de dados somente leitura que fornecem contexto sem computação significativa ou efeitos colaterais. Um recurso pode buscar o conteúdo de um arquivo, recuperar um registro de um banco de dados ou obter documentação. Seu propósito é enriquecer o prompt do LLM com informações relevantes.[6]
2. **Ferramentas (Tools):** Capacidades que permitem ao LLM executar ações ou computações que têm efeitos colaterais. Exemplos incluem chamar uma API externa para obter dados em tempo real, executar um trecho de código ou enviar uma mensagem. As ferramentas capacitam o LLM a interagir com o mundo exterior de forma ativa.[6]

3. **Prompts:** Modelos ou fluxos de trabalho reutilizáveis que guiam as interações entre usuários, modelos de IA e as capacidades disponíveis. Eles podem encapsular lógicas de conversação complexas ou cadeias de raciocínio.[6]

Protocolo de Comunicação

A comunicação dentro do ecossistema MCP é padronizada para garantir a interoperabilidade. O protocolo é construído sobre JSON-RPC 2.0, um protocolo de chamada de procedimento remoto leve.[10] Essa camada de protocolo é transportada por diferentes mecanismos, dependendo do caso de uso [8]:

- **Transporte Stdio:** Usa a entrada/saída padrão para comunicação, ideal para processos locais onde o Host e o Servidor rodam na mesma máquina.[8]
- **Transporte HTTP Transmitível (Streamable HTTP):** Utiliza HTTP com Server-Sent Events (SSE) para mensagens do servidor para o cliente e POST para mensagens do cliente para o servidor, tornando-o adequado para comunicação remota e cenários baseados na web.[8]

1.2. Os Princípios da Arquitetura Hexagonal (Portas e Adaptadores)

A Arquitetura Hexagonal, também conhecida como Arquitetura de Portas e Adaptadores, foi concebida por Alistair Cockburn para criar componentes de aplicação fracamente acoplados.[11] O objetivo principal é isolar a lógica de negócio central (o domínio) das preocupações externas, como interfaces de usuário, bancos de dados, APIs de terceiros ou frameworks de aplicação.[12] Essa separação permite que a aplicação seja "igualmente conduzida por usuários, programas, testes automatizados ou scripts em lote", garantindo que a lógica de negócio possa ser desenvolvida e testada isoladamente de seus dispositivos de tempo de execução.[15]

Princípio 1: Separação Explícita de Preocupações

A aplicação é dividida em três regiões distintas [17]:

- **Lado do Usuário (User-Side) ou Lado de Condução (Driving Side):** É por onde os usuários ou programas externos interagem com a aplicação para iniciar uma ação. Contém o código que permite essas interações, como controladores de API REST, interfaces de linha de comando ou, como veremos, servidores MCP.[17]

- **Lógica de Negócio (Business Logic):** O coração da aplicação, frequentemente visualizado como o "hexágono". Contém o vocabulário de negócio, as regras e a lógica pura que resolvem o problema de domínio específico. Esta parte é completamente agnóstica em relação à tecnologia.[17]
- **Lado do Servidor (Server-Side) ou Lado Conduzido (Driven Side):** Contém a infraestrutura da qual a lógica de negócio depende para cumprir suas tarefas. Isso inclui implementações concretas para persistência de dados, envio de mensagens ou chamadas para outras APIs.[17]

Princípio 2: Inversão de Dependência

As dependências fluem estritamente para dentro. Tanto o Lado do Usuário quanto o Lado do Servidor dependem da Lógica de Negócio. A Lógica de Negócio, por sua vez, não depende de nada externo.[19] Isso é uma aplicação direta do Princípio da Inversão de Dependência do SOLID, onde módulos de alto nível (lógica de negócio) não devem depender de módulos de baixo nível (detalhes de infraestrutura); ambos devem depender de abstrações.[17]

Princípio 3: Isolamento via Portas e Adaptadores

A comunicação através da fronteira do hexágono é mediada por Portas e Adaptadores [11]:

- **Portas (Ports):** São interfaces definidas dentro da lógica de negócio. Elas especificam uma "conversa com propósito" e ditam como a interação pode ocorrer, sem qualquer conhecimento da tecnologia subjacente.[14] Existem dois tipos:
  - **Portas de Condução (Driver Ports / Primary Ports):** A API da aplicação, definindo como atores externos podem conduzir a aplicação.[19]
  - **Portas Conduzidas (Driven Ports / Secondary Ports):** Os requisitos da aplicação, definindo o que a aplicação precisa do mundo exterior.[19]
- **Adaptadores (Adapters):** São implementações concretas das portas e residem fora da lógica de negócio. Eles são responsáveis por traduzir entre o mundo específico da tecnologia (por exemplo, HTTP, SQL) e o mundo agnóstico da tecnologia das portas.[11]

1.3. A Simbiose Natural: Mapeando MCP para o Modelo Hexagonal

A verdadeira força desta arquitetura emerge quando os princípios do MCP e da Arquitetura Hexagonal são combinados. O modelo cliente-servidor do MCP mapeia-se perfeitamente para o lado do condutor de uma Arquitetura Hexagonal. A Arquitetura Hexagonal define um "Lado de Condução", onde atores externos iniciam a interação com o núcleo da aplicação.[1] O MCP, por sua vez, define uma aplicação "Host" (como um IDE alimentado por LLM) que atua como o usuário, iniciando solicitações para obter contexto ou executar ações. Consequentemente, o

Host MCP é um Ator de Condução.[1]

Na Arquitetura Hexagonal, os "Adaptadores de Condução" traduzem solicitações de uma tecnologia específica em chamadas para as "Portas de Condução" da aplicação.[1] A função de um Servidor MCP é receber mensagens JSON-RPC padronizadas e traduzi-las em ações específicas. Portanto, o

Servidor MCP é uma implementação perfeita de um Adaptador de Condução. Ele adapta o protocolo MCP à lógica de negócio central da aplicação. Uma solicitação `tools/call` ou `resources/get` no MCP é um comando direto para a aplicação, que deve ser tratado por uma Porta de Condução.[1]

A implicação mais ampla é profunda: podemos construir um núcleo de aplicação altamente estruturado, testável e focado no domínio, e então simplesmente "conectar" um Servidor MCP como uma das muitas maneiras possíveis de interagir com ele. A lógica de negócio central permanece completamente alheia ao MCP, aos LLMs ou a qualquer tecnologia de IA específica, tornando-a incrivelmente resiliente a mudanças.[1]

Tabela 1: Mapeando Conceitos do MCP para a Arquitetura Hexagonal

Esta tabela serve como uma "Pedra de Roseta" conceitual, fornecendo um mapeamento claro e direto entre as terminologias dos dois padrões, o que é fundamental para a compreensão do restante do relatório.[1]

Conceito MCP	Descrição do MCP	Conceito Hexagonal Correspondente	Descrição Hexagonal

Host	A aplicação de IA voltada para o usuário (ex: IDE, Chatbot) [6]	Ator de Condução (Driving Actor)	A entidade externa que inicia uma interação com a aplicação. [1]
Cliente MCP	Componente dentro do Host que gerencia a conexão [6]	(Parte do Ator de Condução)	O mecanismo que o ator usa para se comunicar.
Servidor MCP	Expõe capacidades (Ferramentas, Recursos) via protocolo MCP [8]	Adaptador de Condução (Driver Adapter)	Um componente específico de tecnologia que traduz solicitações externas em chamadas a uma porta. [1]
Requisição de Ferramenta/Recurso MCP	Uma mensagem JSON-RPC do Cliente para o Servidor [10]	Chamada a uma Porta de Condução	Uma invocação de método em uma interface agnóstica de tecnologia definida pelo núcleo da aplicação.
Núcleo da Aplicação	(Não é um conceito MCP)	O Hexágono	A lógica de negócio e o modelo de domínio isolados e agnósticos de tecnologia. [1]
Banco de Dados/API Externa	Uma fonte de dados ou serviço usado pelo Servidor MCP.	Adaptador Conduzido (Driven Adapter)	Uma implementação específica de tecnologia de uma porta conduzida (ex: um repositório). [1]

Seção 2: Projetando o Núcleo do Hexágono: O Domínio de Negócio AIOps

Esta seção define o coração agnóstico de tecnologia da aplicação. Para nosso MVP, modelaremos um sistema AIOps simplificado que pode consultar dados de incidentes do ServiceNow e dados de log do Splunk, inspirado em casos de uso de integração do mundo real.[21] A definição do domínio permanece idêntica à do blueprint original, pois a lógica de negócio é independente da estrutura física do projeto.

2.1. Identificando Entidades de Domínio e Agregados

O primeiro passo no projeto do núcleo é identificar os objetos de negócio fundamentais. Estes são modelados como entidades e agregados, que encapsulam tanto os dados quanto o comportamento associado.[1]

- **Incidente (Incident):** A entidade central do nosso domínio. Representa um único evento de TI que requer atenção. Conterá atributos essenciais como `id` (identificador único), `shortDescription` (descrição curta), `status` (ex: Novo, Em Andamento, Resolvido), `priority` (prioridade) e `assignmentGroup` (grupo de atribuição). Crucialmente, também conterá uma lista de objetos de valor `WorkNote`, representando as anotações e comentários adicionados ao longo do ciclo de vida do incidente.[1]
- **Nota de Trabalho (WorkNote):** Um objeto de valor que representa uma única entrada de diário associada a um incidente. Contém atributos como `author` (autor), `timestamp` (data e hora) e `text` (o conteúdo da nota). Sendo um objeto de valor, é definido por seus atributos e não possui uma identidade própria separada do Incidente ao qual pertence.[1]
- **Entrada de Log (LogEntry):** Outro objeto de valor, representando uma única linha de log de um sistema como o Splunk. Conterá atributos como `timestamp`, `level` (nível de severidade, ex: INFO, ERROR) e `message` (a mensagem de log).[1]
- **Análise de Incidente (IncidentAnalysis):** Este é um agregado e um conceito de domínio fundamental. Ele representa um pacote contextual completo para a análise de um incidente, combinando a entidade `Incident` com uma lista de `LogEntry`s relevantes. Este agregado encapsula a lógica de negócio de correlacionar um problema (o incidente) com suas evidências (os logs).[1]

2.2. Definindo Regras de Negócio e Invariantes

A lógica de negócio define as regras e restrições que governam o comportamento do domínio. Essas regras, ou invariantes, devem ser sempre verdadeiras e são aplicadas dentro das entidades e serviços de aplicação, completamente isoladas de qualquer framework ou tecnologia externa.[1]

- **Invariante 1:** Uma `IncidentAnalysis` só pode ser gerada para um `Incident` que esteja em um estado 'Ativo' ou 'Novo'. Tentar analisar um incidente já 'Fechado' ou 'Cancelado' resultaria em uma exceção de domínio.[1]
- **Invariante 2:** Uma `WorkNote` só pode ser adicionada a um `Incident` se o conteúdo da nota não for nulo ou vazio.[1]
- **Invariante 3:** A prioridade de um incidente só pode ser escalada, nunca rebaixada, por meio de uma ação de negócio específica.[1]

2.3. Definindo as Portas da Aplicação (A API do Hexágono)

As portas são as interfaces Java que formam o contrato formal entre o núcleo da aplicação e o mundo exterior. Elas são a manifestação do Princípio da Inversão de Dependência, definindo abstrações sobre as quais os componentes externos dependerão.[1]

2.3.1. Portas de Condução (API de Entrada)

Estas interfaces definem o que a aplicação pode fazer. Elas representam a API pública do núcleo do hexágono e serão invocadas pelos nossos adaptadores de condução.[1]

Java

```java
package com.mvp.domain.ports.in;

import com.mvp.domain.model.Incident;
import com.mvp.domain.model.IncidentAnalysis;
import java.util.Optional;

/**
 * Define a API de entrada para o núcleo de gerenciamento de incidentes.
 * Qualquer ator externo (como um Servidor MCP ou um controlador REST)
 * usará esta porta para interagir com a lógica de negócio.
 */
public interface IncidentManagementPort {

    /**
     * Recupera os detalhes de um único incidente.
     * @param incidentId O identificador único do incidente.
```

```
     * @return um Optional contendo o Incidente se encontrado.
     */
    Optional<Incident> getIncidentDetails(String incidentId);

    /**
     * Realiza uma análise de um incidente, correlacionando-o com logs relevantes.
     * @param incidentId O identificador do incidente a ser analisado.
     * @param searchQuery Uma consulta para buscar logs relacionados no sistema de logs.
     * @return um objeto IncidentAnalysis contendo o incidente e os logs correspondentes.
     */
    IncidentAnalysis analyzeIncidentWithLogs(String incidentId, String searchQuery);

    /**
     * Adiciona uma nova nota de trabalho a um incidente existente.
     * @param incidentId O identificador do incidente a ser atualizado.
     * @param workNoteText O texto da nota de trabalho a ser adicionada.
     */
    void addWorkNoteToIncident(String incidentId, String workNoteText);
}
```

2.3.2. Portas Conduzidas (SPI de Saída)

Estas interfaces definem o que a aplicação precisa do mundo exterior para funcionar. Elas são contratos para serviços de infraestrutura.[1]

```
Java

package com.mvp.domain.ports.out;

import com.mvp.domain.model.Incident;
import java.util.Optional;

/**
 * Define o contrato para persistência de Incidentes.
 * A lógica de negócio depende desta porta, mas não de sua implementação (ex: JPA, JDBC).
 */
public interface IncidentRepositoryPort {
    Optional<Incident> findById(String incidentId);
    void save(Incident incident);
}
```

```
Java

package com.mvp.domain.ports.out;

import com.mvp.domain.model.LogEntry;
import java.util.List;
```

```
/**
 * Define o contrato para um serviço de busca de logs.
 * A lógica de negócio depende desta porta, mas não de sua implementação (ex: Splunk, Elasticsearch).
 */
public interface LogSearcherPort {
    List<LogEntry> searchLogs(String query);
}
```

Seção 3: A Estrutura Simplificada do Projeto MVP

Esta seção apresenta a atualização central deste blueprint: a transição de uma estrutura de projeto Maven multi-módulo para uma abordagem de projeto único, mantendo a separação lógica através de pacotes. A principal motivação para esta mudança é a simplificação do processo de build e da configuração do projeto, o que pode acelerar significativamente o desenvolvimento inicial de um MVP.

3.1. De Multi-Módulo para Projeto Único: Justificativa e Consequências para um MVP

A estrutura multi-módulo, conforme delineada no blueprint original, oferece uma separação física robusta das camadas arquitetônicas.[1] Cada módulo (`mvp-domain`, `mvp-application`, `mvp-adapters`) possui seu próprio `pom.xml`, permitindo um controle de dependências rigoroso em tempo de compilação.[22] Por exemplo, é impossível para o código no módulo `mvp-domain` depender acidentalmente de uma biblioteca de framework como o Quarkus, pois essa dependência simplesmente não existiria em seu POM.[24]

Para um MVP, no entanto, essa estrutura pode introduzir uma sobrecarga de configuração. Gerenciar múltiplos POMs, garantir o alinhamento de versões e configurar o reator de build do Maven pode retardar as iterações iniciais.[26] A mudança para um único projeto com um único `pom.xml` simplifica drasticamente o setup inicial, tornando mais fácil para os desenvolvedores começarem a trabalhar e para os pipelines de CI/CD serem configurados.[28]

A consequência mais significativa dessa simplificação é a perda da aplicação de limites arquitetônicos em tempo de compilação. Em um projeto único, um desenvolvedor trabalhando no pacote `com.mvp.domain` poderia, tecnicamente,

adicionar uma declaração de `import` para uma classe do pacote `com.mvp.adapter`. O compilador Java não impediria isso, pois todas as classes estão no mesmo classpath do projeto. Essa violação, se não for controlada, leva ao acoplamento direto entre a lógica de negócio e os detalhes de infraestrutura, corroendo os benefícios fundamentais da Arquitetura Hexagonal e resultando em um "big ball of mud".[13] Essa perda de uma barreira de segurança em tempo de compilação é a principal razão pela qual uma nova forma de fiscalização arquitetônica se torna não apenas recomendada, mas absolutamente crítica.

3.2. Uma Estrutura de Pacotes Concreta para Separação Arquitetônica

Para manter a separação lógica dentro de um projeto único, propomos uma estrutura de pacotes clara e hierárquica que espelha as camadas hexagonais. Essa estrutura de pacotes não é arbitrária; ela se torna a manifestação física da arquitetura lógica e a base para as regras de fiscalização que serão definidas posteriormente.

A estrutura de diretórios dentro de `src/main/java` deve ser a seguinte:

```
com
└── mvp
    ├── domain
    │   ├── model
    │   │   ├── Incident.java
    │   │   ├── WorkNote.java
    │   │   ├── LogEntry.java
    │   │   └── IncidentAnalysis.java
    │   ├── ports
    │   │   ├── in
    │   │   │   └── IncidentManagementPort.java
    │   │   └── out
    │   │       ├── IncidentRepositoryPort.java
    │   │       └── LogSearcherPort.java
    │   └── exception
    │       └── IncidentNotFoundException.java
    │
    ├── application
    │   └── IncidentManagementService.java
    │
    ├── adapter
    │   ├── in
    │   │   └── mcp
    │   │       └── McpServerResource.java
    │   └── out
    │       ├── servicenow
    │       │   ├── ServiceNowIncidentAdapter.java
    │       │   ├── dto
    │       │   │   └── ServiceNowIncidentPayload.java
```

```
│       │   └── mapper
│       │       └── ServiceNowMapper.java
│       └── splunk
│           ├── SplunkLogSearcherAdapter.java
│           └── dto
│               └── SplunkSearchResponse.java
│
└── config
    └── QuarkusApplication.java
```

Essa estrutura cria limites visuais e lógicos claros:

- `com.mvp.domain`: Contém o núcleo do hexágono. É a camada mais interna e pura, sem conhecimento de frameworks ou detalhes de implementação externos.[1]
- `com.mvp.application`: Contém as implementações das portas de entrada (casos de uso). Ela depende do domínio, mas não dos adaptadores.[1]
- `com.mvp.adapter`: Contém todas as implementações específicas de tecnologia, tanto de entrada (`in`) quanto de saída (`out`). Esta é a camada mais externa e depende tanto da aplicação quanto do domínio.[1]

3.3. Insight Crítico: Impondo Limites com ArchUnit

Como a separação de módulos do Maven não está mais presente para impor as regras de dependência, é essencial introduzir uma ferramenta que possa realizar essa verificação. **ArchUnit** é uma biblioteca de teste Java gratuita que permite verificar a arquitetura do seu código.[2] Em vez de confiar na disciplina manual, que é propensa a falhas, transformamos as regras da Arquitetura Hexagonal em testes de unidade executáveis.[29] Esses testes não são opcionais; eles se tornam uma parte fundamental do pipeline de CI/CD, atuando como um portão de qualidade que falha na construção se uma violação arquitetônica for introduzida.[32]

3.3.1. Configurando o ArchUnit

Primeiro, adicione a dependência do ArchUnit para JUnit 5 ao seu `pom.xml`:

XML

```
<dependency>
    <groupId>com.tngtech.archunit</groupId>
    <artifactId>archunit-junit5</artifactId>
    <version>1.2.1</version> <scope>test</scope>
</dependency>
```

Essa única dependência `archunit-junit5` é suficiente, pois ela traz transitivamente as dependências `archunit-junit5-api` e `archunit-junit5-engine` necessárias.[33]

3.3.2. Implementando as Regras Arquitetônicas

Crie uma nova classe de teste em `src/test/java/com/mvp/ArchitectureTest.java`. Esta classe usará as anotações e a API fluente do ArchUnit para definir e verificar as regras de dependência.

Java

```
package com.mvp;

import com.tngtech.archunit.core.importer.ClassFileImporter;
import com.tngtech.archunit.core.importer.ImportOption;
import com.tngtech.archunit.junit.AnalyzeClasses;
import com.tngtech.archunit.junit.ArchTest;
import com.tngtech.archunit.lang.ArchRule;

import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition.noClasses;
import static com.tngtech.archunit.library.Architectures.layeredArchitecture;

@AnalyzeClasses(packages = "com.mvp", importOptions =
ImportOption.DoNotIncludeTests.class)
public class ArchitectureTest {

    // Define os pacotes como constantes para clareza e reutilização
    private static final String DOMAIN_PACKAGE = "..domain..";
    private static final String APPLICATION_PACKAGE = "..application..";
    private static final String ADAPTER_PACKAGE = "..adapter..";

    /**
     * Regra 1: O domínio não deve depender de nada fora dele.
     * Esta é a regra mais fundamental da Arquitetura Hexagonal.
     */
    @ArchTest
    static final ArchRule domain_should_not_depend_on_other_layers =
        noClasses().that().resideInAPackage(DOMAIN_PACKAGE)
            .should().dependOnClassesThat().resideInAnyPackage(APPLICATION_PACKAGE,
ADAPTER_PACKAGE);

    /**
     * Regra 2: A camada de aplicação só pode depender do domínio.
     * Ela não deve conhecer os detalhes dos adaptadores.
     */
    @ArchTest
    static final ArchRule application_should_only_depend_on_domain =
```

```
        noClasses().that().resideInAPackage(APPLICATION_PACKAGE)
            .should().onlyDependOnClassesThat().resideInAnyPackage(DOMAIN_PACKAGE,
"java..", "jakarta..");

    /**
     * Regra 3: Os adaptadores não devem ser acessados pelo núcleo (domínio ou aplicação).
     * Esta regra impede que a lógica de negócio se acople a uma implementação específica.
     */
    @ArchTest
    static final ArchRule adapters_should_not_be_accessed_by_core =
        noClasses().that().resideInAPackage(ADAPTER_PACKAGE)
            .should().onlyBeAccessed().byClassesThat().resideInAPackage(ADAPTER_PACKAGE);

    /**
     * Regra 4: Definição formal de camadas e suas dependências permitidas.
     * Esta é uma maneira mais explícita e poderosa de definir a arquitetura.
     */
    @ArchTest
    static final ArchRule layered_architecture_should_be_respected =
        layeredArchitecture().consideringAllDependencies()
            .layer("Adapters").definedBy(ADAPTER_PACKAGE)
            .layer("Application").definedBy(APPLICATION_PACKAGE)
            .layer("Domain").definedBy(DOMAIN_PACKAGE)

            .whereLayer("Adapters").mayNotBeAccessedByAnyLayer()
            .whereLayer("Application").mayOnlyBeAccessedByLayers("Adapters")
            .whereLayer("Domain").mayOnlyBeAccessedByLayers("Application", "Adapters");
}
```

Explicação das Regras:

- A anotação `@AnalyzeClasses` instrui o ArchUnit a importar e analisar todas as classes do pacote `com.mvp`, excluindo as classes de teste.[36]
- **Regra 1:** Garante que nenhuma classe no pacote `domain` (e seus subpacotes) tenha dependências de classes nos pacotes `application` ou `adapter`. Isso impõe a pureza do domínio.[32]
- **Regra 2:** Assegura que a camada de `application` só pode depender do `domain` e de bibliotecas padrão Java/Jakarta, mas nunca dos `adapter`.
- **Regra 3:** Impede que qualquer classe fora do pacote `adapter` acesse classes dentro dele. Isso garante que os adaptadores sejam detalhes de implementação que não vazam para o núcleo.
- **Regra 4:** Utiliza a API `layeredArchitecture` do ArchUnit para uma definição mais formal e completa das regras de dependência. Ela declara explicitamente as camadas e as regras de acesso permitidas entre elas (por exemplo,

`Application` só pode ser acessada por `Adapters`), fornecendo uma verificação abrangente e de fácil leitura das restrições arquitetônicas.[4]

A integração desses testes no pipeline de CI/CD garante que a arquitetura permaneça limpa e desacoplada, mesmo sem as barreiras físicas dos módulos Maven.

Seção 4: Implementando as Camadas da Aplicação e dos Adaptadores

Esta seção fornece detalhes de implementação concretos para cada camada lógica, agora mapeada para a estrutura de pacotes do projeto único. O código em si é consistente com o blueprint original, mas sua localização e contexto são adaptados à nova estrutura.

4.1. O Serviço de Aplicação (`com.mvp.application`)

A camada de aplicação contém os casos de uso do sistema, implementando as portas de entrada. Ela orquestra as entidades de domínio e delega tarefas de infraestrutura às portas de saída.

`IncidentManagementService.java`

Esta classe implementa a `IncidentManagementPort` e encapsula a lógica de orquestração. Ela é um bean CDI (`@ApplicationScoped`) e injeta as portas de saída (`IncidentRepositoryPort` e `LogSearcherPort`) via construtor, aderindo ao princípio de inversão de dependência.[1]

```
Java

package com.mvp.application;

import com.mvp.domain.model.Incident;
import com.mvp.domain.model.IncidentAnalysis;
import com.mvp.domain.model.LogEntry;
import com.mvp.domain.ports.in.IncidentManagementPort;
import com.mvp.domain.ports.out.IncidentRepositoryPort;
import com.mvp.domain.ports.out.LogSearcherPort;
import com.mvp.domain.exception.IncidentNotFoundException;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import java.util.List;
import java.util.Optional;

@ApplicationScoped
public class IncidentManagementService implements IncidentManagementPort {
```

```
    private final IncidentRepositoryPort incidentRepository;
    private final LogSearcherPort logSearcher;

    @Inject
    public IncidentManagementService(IncidentRepositoryPort incidentRepository,
LogSearcherPort logSearcher) {
        this.incidentRepository = incidentRepository;
        this.logSearcher = logSearcher;
    }

    @Override
    public Optional<Incident> getIncidentDetails(String incidentId) {
        return incidentRepository.findById(incidentId);
    }

    @Override
    public IncidentAnalysis analyzeIncidentWithLogs(String incidentId, String searchQuery) {
        Incident incident = incidentRepository.findById(incidentId)
                .orElseThrow(() -> new IncidentNotFoundException(incidentId));

        // Lógica de negócio: apenas incidentes ativos podem ser analisados.
        if (!"Active".equalsIgnoreCase(incident.getStatus())
&&!"New".equalsIgnoreCase(incident.getStatus())) {
            throw new IllegalStateException("Only active or new incidents can be analyzed.");
        }

        // Constrói a consulta de log com contexto do incidente.
        String contextualQuery = searchQuery + " incident_id=" + incident.getId();
        List<LogEntry> logs = logSearcher.searchLogs(contextualQuery);

        return new IncidentAnalysis(incident, logs);
    }

    @Override
    public void addWorkNoteToIncident(String incidentId, String workNoteText) {
        Incident incident = incidentRepository.findById(incidentId)
                .orElseThrow(() -> new IncidentNotFoundException(incidentId));

        // Lógica de negócio (encapsulada no método da entidade).
        incident.addWorkNote("System", workNoteText);

        incidentRepository.save(incident);
    }
}
```

4.2. O Adaptador de Condução: Servidor MCP (`com.mvp.adapter.in.mcp`)

Este adaptador de entrada traduz as solicitações externas do protocolo MCP em chamadas para as portas de entrada da aplicação.

McpServerResource.java

Implementado como um Recurso JAX-RS do Quarkus, este adaptador atua como o Servidor MCP. Sua única responsabilidade é analisar as solicitações JSON-RPC e invocar os métodos correspondentes na IncidentManagementPort injetada. A exposição via HTTP o torna acessível a Hosts MCP remotos, uma flexibilidade arquitetônica chave.[1]

Java

```
package com.mvp.adapter.in.mcp;

import com.mvp.domain.model.Incident;
import com.mvp.domain.ports.in.IncidentManagementPort;
import jakarta.inject.Inject;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;
import java.util.Optional;

// Representação simplificada de uma requisição MCP
record McpRequest(String method, com.fasterxml.jackson.databind.JsonNode params) {}

@Path("/mcp")
public class McpServerResource {

    @Inject
    IncidentManagementPort incidentManagement;

    @POST
    @Path("/request")
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public Response handleMcpRequest(McpRequest request) {
        // Lógica para analisar a requisição JSON-RPC do MCP
        // e chamar o método apropriado da porta de entrada.
        try {
            switch (request.method()) {
                case "tools/getIncidentDetails":
                    String incidentId = request.params().get("incidentId").asText();
                    Optional<Incident> incident = incidentManagement.getIncidentDetails(incidentId);
                    // Construir e retornar uma resposta JSON-RPC do MCP válida
                    return Response.ok(buildMcpSuccessResponse(incident.orElse(null))).build();

                case "tools/analyzeIncidentWithLogs":
                    String analyzeId = request.params().get("incidentId").asText();
```

```
                String query = request.params().get("searchQuery").asText();
                var analysis = incidentManagement.analyzeIncidentWithLogs(analyzeId, query);
                return Response.ok(buildMcpSuccessResponse(analysis)).build();

            case "tools/addWorkNoteToIncident":
                String noteIncidentId = request.params().get("incidentId").asText();
                String workNoteText = request.params().get("workNoteText").asText();
                incidentManagement.addWorkNoteToIncident(noteIncidentId, workNoteText);
                return Response.ok(buildMcpSuccessResponse("Work note added
successfully.")).build();

            default:
                return Response.status(Response.Status.NOT_FOUND)
                        .entity(buildMcpErrorResponse(-32601, "Method not found"))
                        .build();
        }
    } catch (Exception e) {
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
                .entity(buildMcpErrorResponse(-32603, "Internal error: " + e.getMessage()))
                .build();
    }
}

    // Métodos auxiliares para construir respostas MCP (buildMcpSuccessResponse,
buildMcpErrorResponse)
    private Object buildMcpSuccessResponse(Object result) { /*... implementação... */ return null; }
    private Object buildMcpErrorResponse(int code, String message) { /*... implementação... */
return null; }
}
```

4.3. Os Adaptadores Conduzidos: ServiceNow e Splunk (`com.mvp.adapter.out.*`)

Esses adaptadores de saída implementam as portas de saída, contendo o código específico da tecnologia para interagir com sistemas externos.

4.3.1. Adaptador de Persistência: ServiceNow (`com.mvp.adapter.out.servicenow`)

Este adaptador implementa a `IncidentRepositoryPort` usando o Quarkus REST Client para se comunicar com a Table API do ServiceNow.[38] Ele lida com autenticação e mapeamento de dados entre o modelo de domínio e os DTOs da API externa, utilizando MapStruct para evitar código de mapeamento manual e garantir a compatibilidade com a compilação nativa do Quarkus.[39]

Java

```
package com.mvp.adapter.out.servicenow;

//... imports...
```

```java
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

// Interface do Cliente REST para a API do ServiceNow
@RegisterRestClient(configKey="servicenow-api")
public interface ServiceNowApi {

    @GET
    @Path("/table/incident/{sys_id}")
    @Produces(MediaType.APPLICATION_JSON)
    ServiceNowIncidentPayload getIncident(@HeaderParam("Authorization") String auth,
@PathParam("sys_id") String sysId);

    @PATCH
    @Path("/table/incident/{sys_id}")
    @Consumes(MediaType.APPLICATION_JSON)
    void updateIncident(@HeaderParam("Authorization") String auth, @PathParam("sys_id")
String sysId, ServiceNowUpdatePayload payload);
}

// Implementação do Adaptador
@ApplicationScoped
public class ServiceNowIncidentAdapter implements IncidentRepositoryPort {

    @Inject
    @RestClient
    ServiceNowApi serviceNowApi;

    @Inject
    ServiceNowMapper mapper; // Gerado pelo MapStruct

    @ConfigProperty(name = "servicenow.api.token")
    String apiToken;

    @Override
    public Optional<Incident> findById(String incidentId) {
        ServiceNowIncidentPayload payload = serviceNowApi.getIncident("Basic " + apiToken,
incidentId);
        return Optional.ofNullable(mapper.toDomain(payload));
    }

    @Override
    public void save(Incident incident) {
        ServiceNowUpdatePayload updatePayload = mapper.toUpdatePayload(incident);
        serviceNowApi.updateIncident("Basic " + apiToken, incident.getId(), updatePayload);
    }
}
```

4.3.2. Adaptador de Busca de Logs: Splunk (`com.mvp.adapter.out.splunk`)

Este adaptador implementa a `LogSearcherPort`, utilizando o Quarkus REST Client para interagir com o endpoint `/search/jobs` da API REST do Splunk.[42] Ele encapsula a lógica complexa de criar um trabalho de busca, fazer polling pelo seu status e, finalmente, recuperar os resultados.[44] Uma prática crítica aqui é a construção de consultas SPL (Splunk Processing Language) eficientes, filtrando os dados o mais cedo possível para minimizar a carga de dados e melhorar o desempenho.

Java

```
package com.mvp.adapter.out.splunk;

//... imports...
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import io.smallrye.mutiny.Uni;

@RegisterRestClient(configKey="splunk-api")
public interface SplunkApi {
    @POST
    @Path("/services/search/jobs")
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    @Produces(MediaType.APPLICATION_JSON)
    Uni<SplunkSearchJobResponse> createSearchJob(@HeaderParam("Authorization") String auth, @FormParam("search") String search);

    @GET
    @Path("/services/search/jobs/{sid}")
    @Produces(MediaType.APPLICATION_JSON)
    Uni<SplunkJobStatusResponse> getSearchJobStatus(@HeaderParam("Authorization") String auth, @PathParam("sid") String sid);

    @GET
    @Path("/services/search/jobs/{sid}/results")
    @Produces(MediaType.APPLICATION_JSON)
    Uni<SplunkSearchResponse> getSearchResults(@HeaderParam("Authorization") String auth, @PathParam("sid") String sid, @QueryParam("output_mode") String outputMode);
}


@ApplicationScoped
public class SplunkLogSearcherAdapter implements LogSearcherPort {

    @Inject
    @RestClient
    SplunkApi splunkApi;

    //... outros injects e propriedades de configuração...

    @Override
```

```
    public List<LogEntry> searchLogs(String query) {
        // 1. Criar o job de busca via splunkApi.createSearchJob
        String sid = createSearchJobAndWait(query);

        // 2. Fazer polling do status do job via splunkApi.getSearchJobStatus até que esteja
"DONE"
        //    (usando Mutiny para polling reativo ou um loop síncrono com sleep)

        // 3. Recuperar os resultados via splunkApi.getSearchResults
        SplunkSearchResponse response = splunkApi.getSearchResults("Bearer " + splunkToken,
sid, "json").await().indefinitely();

        // 4. Mapear a resposta para List<LogEntry>
        return mapResponseToLogEntries(response);
    }
    //... métodos auxiliares...
}
```

Tabela 2: Matriz de Responsabilidade de Pacotes e Componentes do MVP

Esta tabela serve como um guia de referência rápida para a estrutura do projeto, esclarecendo a responsabilidade de cada componente chave e sua localização dentro da nova estrutura de pacotes.

Camada	Pacote	Classe/Interface Chave	Responsabilidade
Domínio	com.mvp.domain.model	Incident.java	Representa a entidade de negócio principal. Não contém código de framework.
Domínio	com.mvp.domain.ports.in	IncidentManagementPort.java	Define a API de entrada da aplicação (Porta de Condução).
Domínio	com.mvp.domain.ports.out	IncidentRepositoryPort.java	Define a necessidade de persistência da

			aplicação (Porta Conduzida).
Aplicação	com.mvp.application	IncidentManagementService.java	Implementa os casos de uso orquestrando entidades de domínio e portas conduzidas.
Adaptador (Entrada)	com.mvp.adapter.in.mcp	McpServerResource.java	Implementa um endpoint JAX-RS que atua como um Servidor MCP, adaptando requisições MCP para a IncidentManagementPort.
Adaptador (Saída)	com.mvp.adapter.out.servicenow	ServiceNowIncidentAdapter.java	Implementa IncidentRepositoryPort usando o Quarkus REST Client para chamar a API do ServiceNow.
Adaptador (Saída)	com.mvp.adapter.out.splunk	SplunkLogSearcherAdapter.java	Implementa LogSearcherPort usando o Quarkus REST Client para chamar a API do Splunk.
Configuração	com.mvp.config	QuarkusApplication.java	Ponto de entrada da aplicação Quarkus e local para configurações

			de CDI (ex: Producers).

Seção 5: Padrões de Implementação Avançados e Prontidão para Produção

Esta seção aborda os requisitos não funcionais cruciais para uma aplicação do mundo real, transformando o MVP de um protótipo em um serviço robusto e confiável. A aplicação desses padrões na camada de adaptadores é fundamental para manter a integridade da arquitetura hexagonal.

5.1. Construindo Adaptadores Resilientes com Tolerância a Falhas

Sistemas externos como ServiceNow e Splunk podem falhar, apresentar alta latência ou estar temporariamente indisponíveis.[1] Uma chamada direta de um adaptador pode bloquear threads e causar falhas em cascata. Para mitigar isso, aplicamos padrões de tolerância a falhas diretamente em nossos adaptadores conduzidos, isolando as falhas na borda do nosso hexágono. A solução é usar a extensão

`quarkus-smallrye-fault-tolerance`[45], uma implementação da especificação MicroProfile Fault Tolerance, que permite adicionar resiliência de forma declarativa.

A colocação estratégica dessas anotações é crucial. Elas **nunca** devem ser colocadas nas camadas de domínio ou aplicação. A lógica de negócio no hexágono deve permanecer agnóstica a problemas de infraestrutura.[1] Colocar uma anotação

`@Retry` em um método no `IncidentManagementService` vincularia a lógica de negócio principal a um detalhe de implementação. Ao colocar as anotações no adaptador (por exemplo, `ServiceNowIncidentAdapter`), o `IncidentManagementService` central permanece limpo. Ele simplesmente chama a `IncidentRepositoryPort`, sem saber que a implementação dessa porta está lidando com falhas de forma resiliente.[1]

- `@Retry`: Para lidar com problemas de rede transitórios, podemos anotar os métodos nos adaptadores `ServiceNowIncidentAdapter` e `SplunkLogSearcherAdapter`.[46] Para uma estratégia robusta, podemos configurar `maxRetries`, `delay` e `jitter` para implementar um backoff exponencial, evitando sobrecarregar o serviço externo com novas tentativas imediatas.[46]
- Java

```
// Dentro de ServiceNowIncidentAdapter.java
@Override
@Retry(maxRetries = 3, delay = 200, jitter = 100)
public Optional<Incident> findById(String incidentId) {
    //... chamada ao cliente REST...
}
```

-
-
- `@Fallback`: Para fornecer uma degradação graciosa do serviço quando um sistema externo está completamente indisponível.[48] Por exemplo, se o Splunk estiver fora do ar, o método `searchLogs` no `SplunkLogSearcherAdapter` pode, em vez de lançar uma exceção, retornar uma lista vazia de logs. Isso permite que o `IncidentManagementService` continue seu fluxo, talvez criando uma `IncidentAnalysis` com dados parciais, mas sem travar toda a operação.[49]
- Java

```
// Dentro de SplunkLogSearcherAdapter.java
@Override
@Fallback(fallbackMethod = "fallbackSearchLogs")
public List<LogEntry> searchLogs(String query) {
    //... lógica de busca no Splunk...
}

public List<LogEntry> fallbackSearchLogs(String query) {
    // Loga um aviso sobre a falha e retorna uma lista vazia
    // LOG.warn("Splunk search failed for query: {}. Returning empty list.", query);
    return Collections.emptyList();
}
```

-
-
- `@Timeout`: Para evitar que chamadas de longa duração a APIs externas segurem recursos indefinidamente, podemos usar a anotação `@Timeout` para abortar a operação após um período especificado.[1]
- `@CircuitBreaker`: Para problemas mais persistentes, o padrão Circuit Breaker é ideal. Após um número configurado de falhas consecutivas, o disjuntor "abre", e as chamadas subsequentes falham imediatamente por um período de tempo, sem sequer tentar contatar o serviço externo.[1]

5.2. Uma Estratégia de Teste Multicamadas

A Arquitetura Hexagonal incentiva naturalmente uma pirâmide de testes saudável, onde a maioria dos testes é rápida e focada, com um número decrescente de testes mais lentos e abrangentes.[52]

- **Testes Unitários do Domínio e da Camada de Aplicação:**
  - **Tecnologias:** JUnit 5 e Mockito.[53]
  - **Escopo:** Testar a lógica de negócio pura no pacote `com.mvp.domain` e os serviços de aplicação em `com.mvp.application`.
  - **Implementação:** Em um teste para o `IncidentManagementService`, injetamos o serviço e usamos `@InjectMock` para as interfaces `IncidentRepositoryPort` e `LogSearcherPort`. Isso nos permite simular o comportamento das dependências externas (por exemplo, "quando `findById` for chamado, retorne este `Incident`") e verificar se a lógica de negócio se comporta corretamente em total isolamento, sem iniciar o Quarkus.[53] Esses testes são extremamente rápidos e constituem a base da nossa pirâmide de testes.
- **Testes de Componente/Integração dos Adaptadores:**
  - **Tecnologia:** `@QuarkusTest` para iniciar um ambiente Quarkus em tempo de execução.[56]
  - **Escopo:** Testar cada adaptador individualmente para garantir que ele se comunica corretamente com a tecnologia externa que ele abstrai.
  - **Testando Adaptadores Conduzidos (Driven Adapters):** Para testar o `ServiceNowIncidentAdapter`, precisamos simular a API externa do ServiceNow. Usaremos o **WireMock** para isso. A extensão `quarkus-wiremock` pode gerenciar o ciclo de vida de um servidor WireMock e injetá-lo em nosso teste com `@ConnectWireMock`.[57] No arquivo `application-test.properties`, configuramos a URL do cliente REST para apontar para o servidor WireMock (`quarkus.rest-client.servicenow-api.url=http://localhost:${quarkus.wiremock.devservices.port}`).[38] Isso nos permite definir respostas HTTP esperadas (stubs) e verificar se nosso adaptador faz as requisições HTTP corretas e lida com as respostas adequadamente.
  - **Testando com Dependências Reais (Testcontainers):** Para testar um adaptador de persistência contra um banco de dados real (se aplicável), podemos usar **Testcontainers** com a anotação `@QuarkusTestResource`. Isso iniciará um contêiner de banco de dados (por exemplo, PostgreSQL) para a duração do teste, fornecendo um ambiente de alta fidelidade.[36]

5.3. Arquitetura de Segurança: Protegendo Endpoints e Segredos com Vault

A segurança deve ser projetada na arquitetura desde o início.[1]

5.3.1. Protegendo o Endpoint do Adaptador MCP

O `McpServerResource` é um endpoint exposto e deve ser protegido. Uma abordagem eficaz é implementar um `ContainerRequestFilter` JAX-RS personalizado para validar uma chave de API ou um token de autenticação nos cabeçalhos da requisição.[60] Este filtro deve ser executado com a anotação

`@PreMatching` para se integrar corretamente com o contexto de segurança do Quarkus, garantindo que a verificação de segurança ocorra antes do processamento da requisição.[60]

```
Java

@Provider
@PreMatching
public class ApiKeyAuthFilter implements ContainerRequestFilter {
    //... lógica para extrair o header "X-API-Key" e validá-lo...
    // Se inválido, abortar a requisição com Response.Status.UNAUTHORIZED
}
```

5.3.2. Gerenciamento de Segredos com Vault e Kubernetes

Codificar credenciais para ServiceNow, Splunk ou a chave de API do endpoint MCP no `application.properties` é um risco de segurança inaceitável.[1] A solução é usar um cofre de segredos centralizado.

- **HashiCorp Vault:** Usaremos o Vault como nosso cofre de segredos. A extensão `quarkus-vault` permite que o Quarkus busque propriedades de configuração diretamente do Vault em tempo de execução, em vez de lê-las de arquivos de texto simples.[62]
- **Método de Autenticação Kubernetes:** Para aplicações executadas em Kubernetes, a maneira mais segura e transparente de se autenticar no Vault é o **Método de Autenticação Kubernetes**.[65] A aplicação, em seu Pod, usa seu Token de Conta de Serviço (Service Account Token) projetado para se autenticar no Vault. Isso elimina completamente a necessidade de gerenciar tokens de longa duração ou segredos para a própria aplicação.[67]
- **Política do Vault (Least Privilege):** Definiremos uma política específica do Vault em HCL que concede à identidade da nossa aplicação acesso de leitura (`read`) apenas aos caminhos de segredos específicos de que ela precisa (por

exemplo, `secret/data/mvp/servicenow` e `secret/data/mvp/splunk`). Isso adere estritamente ao princípio do menor privilégio, garantindo que, mesmo que a aplicação seja comprometida, o escopo da exposição de segredos seja minimizado.[70]

Exemplo de política HCL do Vault:

```
Terraform

# Policy para o MVP AIOps
path "secret/data/mvp/servicenow" {
  capabilities = ["read"]
}

path "secret/data/mvp/splunk" {
  capabilities = ["read"]
}
```

5.3.3. Checklist de Implantação de Produção para Kubernetes

Um checklist final, baseado nas melhores práticas de segurança do Kubernetes, garante que a aplicação seja implantada de forma segura [73]:

- **Segurança do Pod:**
  - Executar como um usuário não-root (`runAsNonRoot: true`).[75]
  - Desabilitar a escalada de privilégios (`allowPrivilegeEscalation: false`).[75]
  - Usar um sistema de arquivos raiz somente leitura (`readOnlyRootFilesystem: true`).[75]
- **Segurança da Imagem:**
  - Usar imagens de base mínimas (distroless) para reduzir a superfície de ataque.[75]
  - Escanear imagens em busca de vulnerabilidades usando ferramentas como **Trivy** ou **Snyk** no pipeline de CI/CD.[77]
- **Políticas de Rede (Network Policies):**
  - Implementar recursos `NetworkPolicy` rigorosos para restringir o tráfego de entrada e saída apenas ao que é estritamente necessário (por exemplo, permitir saída apenas para os endpoints da API do ServiceNow e Splunk).[75]
- **Controle de Acesso Baseado em Função (RBAC):**

    - Usar uma `ServiceAccount` dedicada para a aplicação com permissões RBAC mínimas. Evitar o uso da `ServiceAccount` padrão.[81]

Seção 6: Recomendações Estratégicas e Conclusão

Este relatório defende uma abordagem arquitetônica deliberada e baseada em princípios. Ao combinar a camada de contexto de IA padronizada do MCP com a estrutura robusta e desacoplada da Arquitetura Hexagonal, cria-se um sistema que é maior do que a soma de suas partes.[1] A arquitetura proposta, agora simplificada para uma estrutura de projeto único para o MVP e fortalecida com a fiscalização via ArchUnit, não é o caminho de menor resistência para um protótipo simples, mas é o caminho ideal para construir um MVP sério e pronto para produção.

Benefícios Chave Revisitados

- **Manutenibilidade:** A clara separação de preocupações, agora imposta por pacotes e verificada por testes, torna o sistema mais fácil de entender, modificar e manter ao longo do tempo. A lógica de negócio está em um lugar, a lógica de integração em outro.[13]
- **Testabilidade:** A arquitetura permite uma estratégia de teste abrangente e eficiente, resultando em maior qualidade e confiança no software. A capacidade de testar o núcleo de negócios em total isolamento é um benefício imensurável.[82]
- **Flexibilidade e Evolução:** O núcleo de negócio está isolado de mudanças em tecnologias externas. A API do Splunk pode ser substituída por um provedor de logs diferente, o ServiceNow por outra ferramenta de ITSM, ou até mesmo o provedor de LLM pode ser trocado, tudo sem impactar o hexágono. Essa adaptabilidade é essencial em um cenário tecnológico que muda rapidamente.[20]

Recomendação Estratégica e Caminho Evolutivo

A adoção da arquitetura combinada de MCP e Hexagonal, mesmo em um projeto único, representa um investimento inicial em design que se paga exponencialmente ao longo do ciclo de vida do produto. Ela antecipa decisões arquitetônicas críticas, mitigando a dívida técnica e estabelecendo uma base sólida para o crescimento futuro.[1]

A abordagem de projeto único é ideal para MVPs e equipes menores, onde a agilidade e a simplicidade da configuração inicial são primordiais. No entanto, é

crucial reconhecer que esta é uma otimização para a fase inicial. À medida que a aplicação cresce em complexidade ou o tamanho da equipe aumenta, a reintrodução de uma estrutura multi-módulo pode se tornar vantajosa. A migração de volta para o modelo multi-módulo original seria um passo evolutivo natural, facilitado pela estrita separação lógica mantida pelo ArchUnit. As regras do ArchUnit garantem que, quando chegar a hora de dividir o projeto, as dependências já estarão corretas, tornando a refatoração uma tarefa mecânica de mover pacotes para novos módulos Maven, em vez de um complexo esforço de desacoplamento.

Este blueprint atualizado fornece a estrutura necessária para construir uma aplicação de IA que não é apenas funcional, mas também resiliente, segura e construída para durar. Para equipes que buscam construir sistemas de IA de nível empresarial de forma pragmática, este é o caminho recomendado.