INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




NewStatusMonitor - Solução de Telemetria

ANDRÉ FILIPE LAGES NOVO
Junho de 2023

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

NewStatusMonitor - Solução de Telemetria

André Filipe Lages Novo

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientador: Dr. Paulo Gandra de Sousa
Supervisor: João Areias

Porto, 30 de junho de 2023

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 30 de junho de 2023

If we lose credibility just by admitting fault, then we didn't have any in the first place.

ISSHO FUJITORA

Resumo

Monitorizar e extrair valor a partir do comportamento de uma aplicação implantada em ambiente de produção é uma mais valia para qualquer empresa. Através deste processo é possível mitigar falhas e problemas que possam afetar a produtividade da mesma e, consequentemente, a credibilidade da empresa no mercado. Permite também avaliar de que forma é que os utilizadores finais mais comunicam com a aplicação desenvolvida e desta forma, determinar padrões por forma a melhorar a experiência final como um todo. Dito isto, esta tese descreve a construção de uma *pipeline* de observabilidade capaz de capturar, transformar e posteriormente apresentar dados telemétricos através de uma interface gráfica intuitiva, por forma a oferecer as vantagens anteriormente descritas à empresa proponente.

Palavras-chave: Monitorização, Telemetria, Desempenho, Observabilidade

Abstract

Monitoring and extracting value from the behavior of an application deployed in a production environment is an advantage for any company. Through this process, it is possible to mitigate failures and problems that can affect the company's productivity and, consequently, its credibility in the market. It also allows to assess how end users communicate most with the developed application and thus determine patterns in order to improve the final experience as a whole. That said, this thesis describes the construction of an observability pipeline capable of capturing, transforming and subsequently presenting telemetry data through an intuitive graphical user interface in order to offer the previously described advantages to the proposing company.

Agradecimentos

Antes de mais nada, gostaria de agradeceder ao Professor Dr. Paulo Gandra de Sousa por todo suporte e envolvimento manifestados durante o decorrer deste projeto. Aproveito também para agradecer ao meu supervisor, João Areias, pela ajuda e tempo despendido neste trabalho. Gostaria também de expressar o meu mais profundo agradecimento à minha família e amigos, com especial destaque para os meus pais, Madrinha, tia Bina, prima Júlia e ao meu melhor amigo Diogo, que me ajudaram ao longo dos anos a tornar este sonho realidade. Por fim, queria deixar o meu bem haja à Estee Wen que dado o seu envolvimento com o projeto SigNoz, mostrou-se de grande ajuda nesta caminhada.

A realização deste projeto não teria sido possível sem o apoio incansável das pessoas que me são mais próximas. Um sincero e profundo obrigado a todos vós!

Conteúdo

Lista de Figuras

Lista de Código

Lista de Acrónimos

API	*Application Programming Interface*.
CNCF	*Cloud Native Computing Foundation*.
CPU	*Central Processing Unit*.
CSV	*Comma-separated values*.
DOM	*Document Object Model*.
DTO	*Data Transfer Object*.
E2E	*End-to-end*.
FOSS	*Free Open-Soure Software*.
GC	*Garbage Collector*.
gRPC	Google *Remote Procedure Call*.
HTTP	*Hypertext Transfer Protocol*.
HTTPS	*Hypertext Transfer Protocol Secure*.
ISEP	Instituto Superior de Engenharias do Porto.
JSON	*JavaScript Object Notation*.
OOTB	*Out-Of-The-Box*.
OTEL	OpenTelemetry Collector.
OTLP	OpenTelemetry *Protocol*.
PromQL	Prometheus *Querying Language*.
RAM	*Random Access Memory*.
REST	*Representational State Transfer*.
RPS	*Requests Per Second*.
SDK	*Software Development Kit*.
SMTP	*Simple Mail Transfer Protocol*.
SRP	*Single Responsability Principle*.
SSD	*Solid-State Drives*.
SSL	*Secure Sockets Laye*.
TMDEI	Tese de Mestrado do Departamento de Engenharia Informática.

UML *Unified Modeling Language*.
URL *Uniform Resource Locators*.

WSL *Windows Subsystem for Linux*.

Capítulo 1

Introdução

Este capítulo visa apresentar o contexto, problema e objetivos associados à elaboração deste projeto, assim como representar o planeamento efetuado e tarefas realizadas. Contém também uma breve enumeração de metodologias, ferramentas e tecnologias que vão auxiliar o desenvolvimento do projeto. Além disso, descreve também a organização do documento de forma a facilitar a compreensão da estrutura e dos conteúdos do mesmo.

1.1 Contextualização

Este trabalho foi desenvolvido no âmbito da unidade curricular do segundo ano, Tese de Mestrado do Departamento de Engenharia Informática (TMDEI) - área de especialização de Sistemas Computacionais do Instituto Superior de Engenharias do Porto (ISEP).

Esta secção pretende introduzir aspetos importantes que estão inerentes ao desenvolvimento do projeto, como é o caso do conceito de monitorização e a sua importância para a organização em causa. Não obstante, é descrito de forma sumária o modelo de negócio da empresa proponente.

A Ciberbit, com sede em Coimbra, tem como principal atividade o desenvolvimento de *software*. Começou como uma empresa de desenvolvimento de jogos, tendo sido uma das partes envolvidas na produção do jogo Portugal 1111: A Conquista de Soure que foi lançado em abril de 2004 para computador. Posteriormente entrou na área de retalho com o *software* cbRetail, ainda hoje usado pela Tiffosi[1]. Atualmente, apesar de ainda oferecer manutenção à ferramenta cbRetail, o seu foco encontra-se centrado no setor de saúde, com o produto I'MTHOM.

O I'MTHOM é uma plataforma moderna e robusta criada para gerir todas as operações informáticas de uma determinada unidade de saúde. Isto é, o I'MTHOM centraliza todos os diferentes aspetos dos fluxos de trabalho clínicos, administrativos e financeiros, em tempo real, num único sítio de modo a facilitar a rápida tomada de decisão pelas pessoas envolvidas nas diferentes camadas [1].

Ora, aplicações distribuídas, como é o caso do I'MTHOM, sofrem, com alguma frequência de lentidão e/ou falhas devido a problemas de *software* e *hardware*, resultando daqui um fraco aproveitamento dos recursos disponíveis [2]. Uma forma de mitigar a ocorrência de falhas, bem como o tempo requerido para voltar a ter os sistemas operacionais, passa por monitorizar as máquinas e os serviços hospedados nas mesmas. A monitorização de sistemas distribuídos envolve a coleta, interpretação e posterior visualização da informação

[1] https://www.tiffosi.com/

considerada benéfica no que diz respeito às interações entre os diferentes serviços envolvidos no modelo de negócio [3].

Dito isto, dentro da Ciberbit S.A já existe uma ferramenta de monitorização implementada, designada internamente por Status Monitor. Esta solução, contudo, é bastante rudimentar, uma vez que apenas se limita a armazenar numa base de dados relacional os erros que recebe do I'MTHOM através de uma *Representational State Transfer* (REST) *Application Programming Interface* (API). Não realiza qualquer tipo de análise ou processamento sob a informação que recebe. Além desta solução, existe um componente WatchDog[2] que efetua um conjunto de pedidos de *Hypertext Transfer Protocol* (HTTP) a um ou a múltiplos *endpoints* do I'MTHOM e que posteriormente envia o resultado desses pedidos para o Status Monitor que os armazena.

Feita a contextualização deste projeto, de uma forma muito genérica, o que é pretendido é ter um maior grau de observabilidade do I'MTHOM. Ou seja, que se consiga ter acesso a mais dados sobre o que está a acontecer na aplicação, nomeadamente, métricas sobre o estado dos vários componentes e, no limite, ter informação sobre todas as interações na aplicação - *traces*.

1.2 Problema

Como referido no subcapítulo anterior, a Ciberbit S.A. tem como principal produto o I'MTHOM, uma aplicação distribuída. No caso desta ficar inoperacional durante longos períodos de tempo, isso pode implicar consequências legais para a empresa, bem como afetar a credibilidade da mesma perante os seus atuais e futuros clientes.

Para evitar tais encargos, a equipa responsável pelas operações *DevOps*[3] necessita de ser capaz de obter informações num espaço de tempo razoável sobre os serviços que precisa manter bem como métricas específicas desses mesmos serviços para assim poder realizar o seu trabalho nas melhores condições. A equipa de desenvolvimento do I'MTHOM também pode fazer uso da informação disponibilizada por esta solução para saber quais partes podem requerer um foco de atenção mais urgente.

Pretende-se, por isso, criar uma *pipeline* de observabilidade que seja capaz de, por exemplo, expandir o atual conjunto de dados que são capturados, caracterizá-los e indexá-los por forma a melhorar o desempenho das futuras operações de consulta.

Dito isto e após uma análise ao produto Status Monitor, foram enumerados alguns problemas cruciais que devem ser resolvidos pela solução descrita neste documento:

- Falta de recolha de métricas importantes para avaliar o estado atual dos diferentes recursos, ou seja, fraca instrumentação;
- Fraca caracterização dos dados armazenados. O Status Monitor apenas armazena erros durante a execução das instâncias do I'MTHOM sem qualquer tipo de processamento ou análise envolvida;
- Fraco desempenho no momento de realizar consultas aos dados armazenados.

[2] https://techdocs.broadcom.com/us/en/symantec-security-software/identity-security/privileged-access-manager-server-control/14-1/reference/utilities/services-and-daemons-in-detail/watchdog-service.html

[3] https://aws.amazon.com/devops/what-is-devops/

1.3 Objetivos

O grande objetivo deste projeto é resolver as limitações existentes da solução existente, no que diz respeito à falta de observabilidade do I'MTHOM que esta disponibiliza.

Neste âmbito foram identificados dois caminhos possíveis para aumentar o grau de observabilidade. Um dos caminhos consiste em desenvolver uma nova *pipeline* de observabilidade de raiz, utilizando ferramentas de uso livre e de código aberto. Outro caminho passa por complementar a solução já existente dentro da empresa com novas funcionalidades e, desta forma, transformá-la assim numa solução suficientemente robusta capaz de atender às necessidades da Ciberbit. Para este projeto, a primeira abordagem foi selecionada, enquanto a segunda foi escolhida para um outro aluno de mestrado. Assim, e da análise da secção anterior, identificaram-se os principais objetivos que esta nova solução deve atender:

- Ser capaz de capturar métricas relativas à infraestrutura que suporta o I'MTHOM, como o uso de *Central Processing Unit* (CPU) e *Random Access Memory* (RAM), entre outras;
- Ser capaz de capturar métricas e *traces* do I'MTHOM, como por exemplo, o número de pedidos por segundo e a duração média dos mesmos;
- Ser capaz de capturar métricas da base de dados que suporta o modelo de negócio do I'MTHOM, como mas não só, o número de transações por segundo ou as consultas que executam com maior frequência;
- Ser capaz de armanezar os dados telemétricos capturados pelos componentes supracitados eficientemente;
- Ser capaz de apresentar os dados telemétricos capturados e processados numa interface gráfica central;
- Ser capaz de criar regras de alertas com base nos dados telemétricos capturados;
- Uma interface *web* que permita a visualização e gestão de toda a informação capturada.

O cumprimento dos objetivos acima enumerados permite à empresa proponente garantir que o seu produto de gestão clínica seja monitorizado e possua um bom grau de observabilidade, por forma a providenciar à equipa de DevOps dados analíticos claros, coerentes e fiáveis relativamente ao estado atual e passado dos sistemas informáticos que sustentam o seu modelo de negócio, oferecendo-lhes assim uma forma mais adequada de realizar o processo de manutenção dos mesmos. Além disso, permite também auxiliar a equipa de desenvolvimento do I'MTHOM com informações relevantes, ao ser capaz de registar e analisar o comportamento dos utilizadores perante o produto em questão ao longo do tempo e, desta forma, ajudar a identificar possíveis pontos a melhorar.

1.4 Abordagem

Nesta secção é apresentada a abordagem adotada por forma a permitir atingir os objetivos anteriormente definidos:

- Inicialmente é feita uma investigação dos conceitos referentes ao contexto do projeto, ou seja, relativos às temáticas de monitorização e telemetria, por forma a se compreender melhor os objetivos definidos. Ainda nesta fase são também exploradas

algumas soluções já existentes que podem ser enquadradas na construção da *pipeline* final, tendo por base o estudo da solução existente e as limitações identificadas da mesma;

- Após a concretização da primeira fase, procede-se ao levantamentos dos requisitos funcionais e atributos de qualidade exigidos pela solução e após isso, ao desenho da *pipeline* de observabilidade com recurso à *Unified Modeling Language* (UML);
- Com a segunda fase realizada, segue-se para a implantação da solução propriamente dita com posterior validação e avaliação das capacidades da mesma. No final, com espírito crítico, é feita uma revisão geral refletindo o estado final do projeto e possíveis pontos a melhorar.

1.5 Planeamento

Para o planeamento deste projeto, procedeu-se a diversas reuniões de acompanhamento com o supervisor da organização - que neste projeto ocupa também a posição de *stakeholder*-, tanto inicialmente, para delinear os objetivos a serem concretizados como posteriormente para proceder à verificação dos desenvolvimentos efetuados e acompanhamento da concretização das tarefas determinadas.

Durante o desenvolvimento do projeto, adotou-se a metodologia SCRUM. Esta metodologia tem por base a filosofia AGILE e define um conjunto de valores, princípios e práticas de modo a criar uma estrutura de trabalho que ajuda as equipas a gerir o seu trabalho de uma forma simples, rapída e eficaz. Através disso é capaz de ajudar as mesmas a entregarem valor (neste caso, em formate de funcionalidades de *software*) aos clientes de forma contínua [4].

A cada iteração concretizada, o supervisor forneceu o seu *feedback*, seguido de definição de novas tarefas a realizar, contemplando os seus requisitos e a respetiva prioridade das mesmas.

1.6 Estrutura do documento

O presente relatório encontra-se dividido em mais 6 (seis) capítulos:

- Estado da arte: onde a literatura relativa a este projeto é apresentada e discutida;
- Análise: onde é feita a análise da solução em vigor, bem como um levantamento dos requisitos funcionais e atributos de qualidade;
- Desenho: onde é feito o desenho arquitetural da *pipeline*;
- Implementação: onde a implementação e implantação da *pipeline* é abordada;
- Avaliação: onde é realizada a avaliação da solução com a apresentação e posterior de resultados;
- Conclusão: onde é feita uma passagem geral pelo projeto desenvolvido, mencionando os objetivos concluídos e o trabalho futuro da solução.

Capítulo 2

Estado da arte

Neste capítulo é apresentado o atual estado da arte sobre o contexto no qual este projeto se insere. Desta forma, inicialmente é realizado um enquadramento teórico sobre os principais conceitos que esta solução aborda, nomeadamente: telemetria, observabilidade, métricas, *logs* e *distributed traces*. É apresentada posteriormente uma análise das solução existentes.

2.1 Enquadramento teórico

Desde o *e-commerce* às redes sociais, os sistemas distribuídos permitiram o avanço da ciência nas mais variadas áreas, ao mesmo tempo que ajudaram a melhorar a qualidade de vida de milhões de pessoas. Contudo, a tarefa de diagnosticar problemas nestes sistemas pode ser bastante desafiante. Estudos revelaram que os programadores gastam quase 50% do seu tempo útil em operações de *debugging* [5].

Ora, podemos caracterizar um sistema distribuído como sendo a coleção de múltiplos serviços a trabalhar em conjunto para atender um determinado resultado. Cada serviço, individualmente, é um programa determinístico capaz de executar de forma independente e concorrente com os outros serviços. Dado o método de funcionamento dos mesmos, existem algumas razões que justificam a razão das operações de depuração, testes e avaliação serem mais difíceis de levar a cabo neste tipo de sistemas do que em sistemas tradicionais (sequenciais) [3]:

- Um sistema distribuído tem múltiplos pontos de controlo. Por conseguinte, técnicas/abordagens de monitorização e depuração de sistemas sequenciais, como o rastreamento de operações em tempo de execução com recurso a *breakpoints*, baseados em contadores e registo dos múltiplos estados do processos, precisam de ser amplificadas de modo a se tornarem passíveis de serem utilizadas no contexto de sistemas distribuídos;
- Os atrasos nas comunicações entre os diferentes *nodes* que compõe um sistema distribuído tornam o processo de determinar o estado do sistema (como um todo) num determinado ponto do tempo uma tarefa complexa. Por exemplo, a inicialização de uma tentativa para determinar este estado deve ser realizado a partir de um *node* e posteriormente os restantes são notificados mais tarde sobre esta tentativa, sem qualquer capacidade de se prever quando especificamente;
- Sistemas distribuídos assíncronos são inerentemente não-determinísticos. Isto significa que dois pedidos realizados a um sistema deste tipo podem produzir diferentes

sequências de eventos internamente, mas não significando que ambas não sejam válidas. Ora, isto torna difícil reproduzir erros e testar situações passíveis de acontecer, mas improváveis;

- Ao contrário do que acontece num sistema sequencial, monitorizar um sistema distribuído altera o seu comportamento. O comportamento de um programa sequencial não é afetado pelo tempo que passa entre duas instruções de execução consecutivas, por exemplo, um *symbolic debugger* pode interromper a execução de um processo sequencial num determinado *breakpoint* sem que isso afete as posteriores instruções de execução do processo. Por outro lado, num sistema distribuído, parar ou reduzir o desempenho de um processo pode resultar na alteração do comportamento de todo o sistema.

Tendo por base os pontos supracitados, é de se notar que à medida que estes sistemas crescem, os mesmos tornam-se cada vez mais complexos. Entender e analisar o comportamento das diferentes operações que acontecem nestes sistemas com um grau suficiente para se diagnosticar problemas também se torna, evidentemente, mais difícil. Por exemplo, em sistemas de larga escala, as pessoas responsáveis, tipicamente, tendem a não saber a 100% quais aplicações estão realmente a executar nos seus sistemas, e como resultado, fraude, desperdício e abuso de recursos computacionais tem crescido ao ponto de se tornar um sério problema para as empresas [6].

2.1.1 Telemetria

De forma a ajudar a diagnosticar problemas deste tipo, engenheiros de diferentes áreas e programadores vêm a adicionar instrumentação, tipicamente, a todas as camadas e a todos componentes que compõem o sistema. A informação recolhida através desta instrumentalização é continuamente captada em tempo de execução e pode ser dividida em 3 (três) grandes tipos ou pilares: *time series metrics*, *logs traces*. Estes são descritos com maior pormenor nas subsecções 2.1.2, 2.1.3 e 2.1.4, respetivamente. Além destes, outros tipos de dados podem também ser recolhidos, como por exemplo, *core dumps*. Contudo, estes dados, tipicamente, não são capturados continuamente porque os benefícios podem nem sempre compensar o custo da operação de o fazer. Isto é, a recolha dos mesmos pode implicar uma redução de desempenho do sistema alvo ou estes não são recolhidos em volumes suficientemente elevados para serem posteriormente alvos de operações de análise automatizadas [7].

Dito isto, telemetria é o conjunto de informação utilizada para se conseguir monitorizar as aplicações de interesse ao modelo de negócio. É um termo que cobre tudo desde *logs*, *time series metrics* até dados de desempenho recentemente definidos, como os *distributed traces*. O termo telemetria existe desde o começo da Internet e num contexto mais generalizado, pode-se dizer que é mais antigo que o próprio conceito de *software* e vai até ao começo da instrumentalização de aparelhos eletrónicos. Hoje em dia, em termos de telemetria de aplicações informáticas, adicionar instrumentação a um produto - quer o *source code* mude ou apenas as configurações - é o mecanismo como os dados telemétricos são gerados. Ora, enquanto os conceitos de métricas e de *logs* já existem há algum tempo na indústria, os *distributed traces* apenas ganharam relevância quando as organizações começaram a adotar micro-serviços e outras arquiteturas de *software* sem qualquer tipo de acoplamento.

O termo telemetria está também relacionado com o termo observabilidade, apesar de representarem coisas diferentes. Enquanto o termo telemetria possui já uma longa história,

no contexto de *software*, observabilidade refere-se à habilidade de se ser capaz de conectar a causa ao efeito num sistema distribuído. Ferramentas de observabilidade ajudam as empresas a entender quais as alterações internas que levaram a uma mudança do desempenho visível para o utilizador final. O termo observabilidade é, geralmente, colocado como um sinónimo de telemetria, o que não é bem verdade. O termo observabilidade, diz respeito ao processo de se ser capaz de obter valor prático dos dados obtidos através de telemetria. Assim, enquanto a telemetria (seja na forma de métricas, *logs* e *traces*) é uma parte necessária da observabilidade, não é suficiente: é necessário haverem ferramentas capazes de analisar e extrair valor desses dados e posteriormente fornecer uma visualização clara dos mesmos num formato legível e prático para auxiliar na tomada de decisões [8].

Ora, geralmente, dados de instrumentalização através da rede podem ser obtidos com recurso a subscrições (*subscriptions* - *push*) e consultas (*queries* - *pull*). Uma subscrição, em termos gerais, é um contrato entre um produtor (*publisher*) e um consumidor (*subscriber*). Após a configuração inicial, o produtor envia os dados produzidos a todos os subscritores interessados até que o período estipulado no contrato (*subscription*) expire.

Por outro lado, consultas (*queries*) são utilizados quando um consumidor necessita prontamente de uma resposta. A informação requisitada pode ser extraída diretamente de um armazém de dados específico ou sintetizada e processada a partir de dados brutos (*raw data*).

De forma sucinta, os dados podem ser consultados quando necessários (*queries*), mas na maioria dos casos, subscrever a canais de dados (*subscriptions*) é mais eficiente e é capaz de reduzir a latência de modo a que o utilizador final não sinta qualquer quebra de desempenho. Do ponto de vista de uma aplicação consumidora deste tipo de dados, existem quatro tipos principais de dados telemétricos que podem ser subscritos ou consultados [9]:

- Dados simples (*Simple Data*): dados que estão constantemente disponíveis a partir de um armazém de dados num dispositivo específico dentro da rede;
- Dados derivados (*Derived data*): dados que após serem recolhidos a partir da rede precisam de ser sintetizadas ou processados. As operações de processamento deste tipo de dados podem ser implementadas de forma estática ou dinâmica nos diferentes dispositivos da rede;
- Dados provenientes de eventos (*Event-triggered Data*): dados que são adquiridos mediante a ocorrência de determinados eventos. Um exemplo deste tipo de dados é a recolha do estado operacional de um componente (*up* e *down*);
- Dados de fluxos contínuos (*Streaming Data*): dados que são gerados continuanente. Podem se tratar de *time series metrics* ou *core dumps*, por exemplo. Este mecanismo tem a vantagem de conseguir refletir o estado atual da rede em tempo real, contudo requer um enorme consumo de largura de banda (*bandwidth*) e de poder de processamento. Dito isto, abordagem de recolha deste tipo de dados passa, tipicamente, por canais de subscrições (*push*);

Os dados telemétricos acima enumerados não são mutuamente exclusivos. Aliás, eles são geralmente mistos e interligam-se uns aos outros. Dados derivados são formados através do agrupamento de uma série de dados simples. Por sua vez, dados provenientes de eventos podem ser de natureza simples ou derivada e os dados de fluxo contínuo podem ser provenientes de eventos recorrentes do sistema [9]. As relações entre estes tipos de dados encontram-se ilustradas na Figura 2.1.

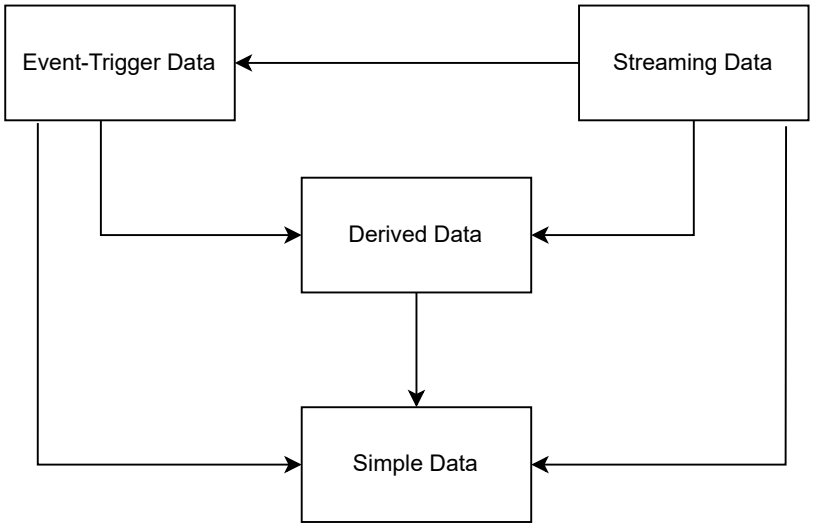

Figura 2.1: Interligação entre os diferentes tipos de dados telemétricos [9].

Dito isto, o foco da nosa *pipeline* centra-se em duas fontes de dados para posteriormente conduzirmos a nossa análise: dados numéricos, que são tipicamente recolhidos do sistema operativo ou do próprio *hardware* e *traces* capturados do *I'MTHOM*, uma aplicação distribuída. A partir destes *traces* é depois também possível extrair dados numéricos.

2.1.2 Métricas

Métricas (*Metrics*) são estatísticas acerca do desempenho de um componente específico capturas e armazenadas tendo por base um determinado ponto no tempo ((*time series metrics*)), cada métrica, tipicamente, representa o contador de algum recurso e é armazenada juntamente com o *timestamp* associado [10]. Por exemplo, considerando o produto de *software I'MTHOM* como exemplo, o registo da latência média dos pedidos feitos à mesma ao longo de um determinado período de tempo é um exemplo de uma métrica. Este tipo de estatísticas são uma mais valia para detetar e analisar a mudança de comportamento de uma solução informática ao longo do tempo. Além disso, as métricas, dada a sua natureza, são facilmente mantidas mesmo quando a aplicação escala, uma vez que, ainda dentro deste mesmo exemplo, o registo da latência média ocupa o mesmo espaço em disco independentemente da nossa aplicação processar 10 (dez) pedidos por segundo ou dez milhões.

Ora, hoje em dia todos os componentes de todas as camadas que formam a *software stack* de uma solução moderna contém dados numéricos passíveis de serem captados. A camada física, por exemplo, é capaz de fornecer métricas relativas ao consumo de energia e à temperatura de diferentes componentes elétricos através do uso de sensores físicos integrados, a camada de rede, por sua vez, é capaz de registar todas as estatísticas que envolvem os pacotes de dados trocados e posteriormente tanto o sistema operativo como os próprios *middlewares* das aplicações contém registos de uso e estatísticas de desempenho detalhadas para cada um dos múltiplos serviços do sistema.

Geralmente, em um sistema distribuído de larga escala, este tipo de informação é agrupada em blocos de dados e armazenada para uso futuro. Daqui resulta um fluxo de *time series data* provenientes de cada *node* do sistema. É comum serem registadas centenas a milhares de diferentes métricas por *node* [11].

Contudo, é de realçar que existem restrições na quantidade de dados de instrumentação que podem ser recolhidos por um sistema. Capturar métricas relativas ao desempenho de todos os CPUs ao mesmo tempo, tipicamente, não é possível devido a restrições de *hardware*. Além disto, os dados capturados precisam posteriormente de ser guardados e o armazenamento que a empresa tem dedicado a operações de telemetria pode ser limitado. Outro ponto importante também é que os dados telemétricos necessitam, geralmente, de serem transmitidos através da rede, consumindo desta forma largura de banda *bandwidth*. Não obstante, certos processos de captura de métricas precisam de ser realizados na mesma *thread* e/ou no mesmo processo onde o componente da aplicação que se deseja monitorizar está a executar, podendo causar assim uma redução no desempenho da aplicação.

Devido aos múltiplos fatores enumerados, as métricas que se querem capturar e analisar devem ser alvos de um rigoroso estudo previamente para se tentar mitigar ao máximo os constrangimentos de desempenho. Tanto para aproveitar ao máximo os recursos disponíveis como para posteriormente ser possível tomar decisões com base nas melhores informações disponíveis. Esta abordagem torna-se cada vez mais verdade tendo por base as aplicações de *software* modernas, onde a mudança é constante, seja no comportamento do utilizador, na infraestrutura da empresa ou na própria aplicação informática [12].

2.1.3 Logs

No contexto de engenharia informática, podemos definir *logs* como sendo a coleção de eventos que é capturada pelo sistema com metadados associados. Estes metadados numa aplicação *web*, tipicamente, contêm mensagens (*log messages*) com informação relativa ao estado do sistema, ao processo (ou *thread*) e o identificador único do pedido *web* onde este evento foi criado/lançado, entre outras informações [13].

Comecemos, então, por definir do que se trata um evento neste contexto específico. No contexto de uma aplicação *web*, um evento é um registo de tudo o que aconteceu dentro de um pedido específico que interagiu com a aplicação. Para criar um registo deste evento (*log*), procede-se à criação de um mapa vazio no início, bem quando um pacote de dados chega através da rede ao nosso serviço. Durante o período de tempo de vida desse pedido, qualquer detalhe que possa ser considerado interessante sobre o que está acontecer - identificadores únicos, valores de variáveis, valores de cabeçalhos (do termo inglês *headers*), todos os parâmetros passados através do pedido, o tempo de execução, etc. - deve ser adicionado a esse mesmo mapa. Posteriormente, quando o processamento desse pedido chega ao fim seja através de um fluxo de sucesso ou de insucesso, todo o mapa é capturado e guardado. Os dados escritos neste mapa são, tipicamente, organizados e formatados, como pares de chave-valor (do inglês *key-value pairs*), de modo a facilitar as futuras operações de pesquisa [13].

Ao se efetuar este tipo de registos, facilita-se a identificação de erros, avisos ou condições anormais que possam acontecer durante a execução do código, permitindo assim que tanto os administradores dos sistema como os próprios desenvolvedores conseguiram ter um maior grau de observabilidade das atividades passadas que o sistema realizou e, desta forma, reduzir drasticamente o tempo despendido na identificação e resolução de problemas complexos que possam existir [13].

2.1.4 Distributed traces

Distributed traces são o mais novo tipo de dados telemétricos a ser integrado como parte das aplicações de monitorização. De certo modo, *traces* acabam por ser um tipo especial de *logs* que seguem uma estrutura diferente: um *distributed trace* consiste numa série de eventos relativa a um único pedido de um utilizador que acaba por ser tratado por diferentes serviços. Em outras palavras, *traces* são *logs* embutidos com um determinado *index* de modo a ajudar a descobrir e rastrear uma transação distribuída no tempo [14].

Ao nível de aplicações distribuídas, *traces* - neste contexto, *distributed traces* - são mais úteis que *logs* específicas de cada *node*, uma vez que estas disponibilizam informação ao nível do *node* como um todo e um único *node* no contexto de um sistema distribuído pode estar a hospedar múltiplos serviços e certos problemas que acontecem ao nível da camada aplicacional podem não se refletir posteriormente ao nível do *node* como um todo. Um dos maiores desafios de se usar *distributed traces* para diagnosticar problemas deste tipo é a dificuldade em saber quais valores recolher, uma vez que a operação de *tracing* pode provocar constrangimentos de desempenho significativos [14].

O rastreamento distribuído (*distributed tracing*) é uma mais valia para conseguir obter *End-to-end* (E2E) visibilidade em sistemas distribuídos. O mesmo é útil para diagnosticar uma série de problemas de desempenhos em sistemas de produção, uma vez que os *traces* registam tanto os eventos que aconteceram internamente durante o processamento de um determinado pedido, como a ordem de execução destes (mesmo que concorrentemente) e combinam toda esta informação por todos os componentes que interagiram de alguma forma com o pedido. Um objetivo chave quando se está a realizar o desempenho de uma ferramenta focada em *distributed tracing* é que a mesma seja capaz de realizar o rastreamento em sistemas de produção, isto é, ser capaz de produzir, recolher e armazenar dados de instrumentação de grande volume, tendo um custo monetário razoável. Num cenário extremo/limite, este tipo de ferramentas podem capturar todo o processamento E2E para todos os pedidos que o sistema de produção opere. Ora, um único pedido pode produzir uma quantidade de dados significativa e capturar todos estes dados, como: os eventos que ocorrem em todos os processos e máquinas ao longo do fluxo de processamento, a ordem dos mesmos (tendo em consideração a execução concorrente de uma subparte destes), as dependências que estes possuem entre si, toda a informação temporal envolvida, entre outros, torna-se insuportável para qualquer sistema [15].

Tendo este cenário em mente, a abordagem, tipicamente, utilizada para mitigar o uso excessivo de recursos é a criação de conjuntos de amostras, do inglês *sampling*. Em vez de se rastrear todo e qualquer pedido recebido, opta-se por apenas recolher e persistir *traces* para um pequeno conjunto de pedidos. De modo a garantir que os dados capturados para este pequeno grupo de pedidos são efetivamente úteis, as decisões tomadas para um pedido devem ser aplicadas a todos. Por exemplo, ou se opta por capturar e persistir todos os *traces* registados por todos os pedidos ou simplesmente se abdica deste registo em todo o grupo de amostragem. Em certos casos práticos, a estratégia de *sampling* é passível de ser implementada usando apenas 0.1% de todo o universo de pedidos recebidos [16]. Esta é a abordagem que se pondera utilizar aquando do desenvolvimento da solução descrita neste documento.

2.2 Soluções existentes

Como já referido, um dos problemas mais comuns na monitorização de sistemas distribuídos bem como de sistemas sequenciais é fazer com que a enorme quantidade de dados capturada resulte, efetivamente, em dados legíveis para as partes interessadas. Ora, uma *pipeline* de observabilidade trata-se de um processo que envolve não só a captura de dados telemétricos, no contexto deste projeto métricas e *traces*, mas também o envio dos mesmos para uma ferramenta de análise onde após receberem o devido tratamento, são posteriormente apresentados através de uma interface gráfica, geralmente *web*. Tendo por base esta premissa, nesta secção são avaliadas algumas das soluções existentes passíveis de serem utilizadas na construção da *pipeline* pretendida, onde para além de se procurar desacoplar a ferramenta de monitorização/telemetria da ferramenta de análise, também apenas se optou por avaliar soluções gratuitas e cujo código fonte fosse disponível ao público por forma a evitar o *vendor lock-in*[1]. Dito isto, estas vão ser as soluções analisadas:

- OpenTelemetry;
- Prometheus;
- Jaeger;
- SigNoz.

É de realçar que as soluções de análise selecionadas - as 3 (três) últimas supracitadas - foram selecionadas tendo em vista o foco deste projeto. Apesar de se propor a ser uma *pipeline* de observabilidade, o mesmo foca-se essencialmente em *traces* e métricas, tendo em consideração os objetivos sugeridos pela empresa proponente.

2.2.1 OpenTelemetry

O OpenTelemetry foi anunciado formalmente em Maio de 2019 como sendo a próxima versão do OpenTracing e do OpenCensus. Estes dois produtos tinham propósitos similares, mas diferentes abordagens para os atingir. Devido à necessidade então de haver um padrão uniforme e cedendo à pressão da comunidade, uma pequena equipa técnica foi formada com o objetivo de criar um protótipo de uma API mista de ambos os serviços. Este protótipo foi então apelidado de OpenTelemetry. O objetivo principal do OpenTelemetry é, então, ser capaz de fornecer um conjunto de APIs e *Software Development Kit* (SDK)s pronto a usado para capturar *logs*, métricas e *traces* [17].

Com este produto, é possível realizar instrumentação manual e automática. A instrumentação manual pressupõe que o código da aplicação alvo tenha que receber novos blocos de código, enquanto que a instrumentação automática faz uso de agentes inteligentes que executam em paralelo com a aplicação alvo e são capazes de captar métricas desta. Apesar de a instrumentação automática ser a mais simples de incorporar, esta, como seria de se esperar, não possui a mesma flexibilidade da instrumentação manual, uma vez que para captar dados personalizados do modelo de negócio em que a aplicação alvo se enquadra é preciso, também, haverem blocos de código personalizados para este efeito. Claro está que ambas não são mutuamente exclusivas e podem ser integradas em conjunto na solução [18].

Ora, o OpenTelemetry define determinadas diretrizes que devem ser seguidos pela APIs e pelos SDKs, bem como pela estrutura dos dados propagados e do protocolo utilizado

[1] https://www.cloudflare.com/learning/cloud/what-is-vendor-lock-in/

para o transporte. Para cada linguagem é providenciado uma única API e um único SDK com o qual é possível instrumentalizar a nossa aplicação para gerar dados telemétricos. A *Standard* API, da qual as APIs específicas para cada linguagem derivam, permite que não seja necessário existir alterações no código quando houver necessidade de mudar de SDK. O SDK é quem trata do empacotamento dos dados, da propagação do contexto e do processamento necessário para depois os dados poderem ser exportados, tipicamente, para um OpenTelemetry Collector [19].

Ora, o OpenTelemetry Collector (OTEL) é capaz de capturar dados telemétricos enviados pelos SDKs bem como de outras fontes não inclusas de forma nativa no ecossistema da OpenTelemetry. É também capaz de exportar, posteriormente, estes mesmos dados telemétricos para uma das várias *backends* suportadas, como o Prometheus[2] ou o Jaeger[3]. O OpenTelemetry Collector é capaz de servir tanto como um agente local, ou seja, correr no mesmo ambiente físico da aplicação-alvo, como sendo um serviço central capaz de captar e armazenar dados telemétricos de múltiplos nós [19].

Dito isto, o último componente deste fluxo são os Exporters. Estes permitem a uniformização dos dados num formato de saída comum - cada Exporter permite a conversão/uniformização para um determinado formato de dados - para que os mesmos possam então ser posteriormente transmitidos. O fluxo do OpenTelemetry possui Exporters em dois pontos, nos SDKs que permitem a saída uniformizada de dados dos mesmos para o OpenTelemetry Collector e posteriormente no próprio Collector para permitir a exportação dos dados para um serviço de *backend* à escolha.

Como mencionado anteriormente, o OpenTelemetry define um protocolo independente de empresas e de ferramentas chamado de OpenTelemetry *Protocol* (OTLP). Este protocolo serve para transmitir *logs*, métricas e *traces*. Tendo isto em vista, substituir a ferramenta de análise em uso (*backend*) é tão fácil como alterar a configuração do OpenTelemetry Collector. O OTLP pode ser utilizado para transmitir dados telemétricos do SDK para o Collector, bem como referido antes, do próprio Collector para a ferramenta de análise selecionada. A especificação do OTLP define a estrutura, o tipo de transporte e o mecanismo de entrega para os dados, sendo, por isso, uma escolha segura para o futuro [17].

Dito isto, segue-se agora na Figura 2.2, uma breve representação arquitetural do OTEL para sumarizar o conteúdo até aqui apresentado:

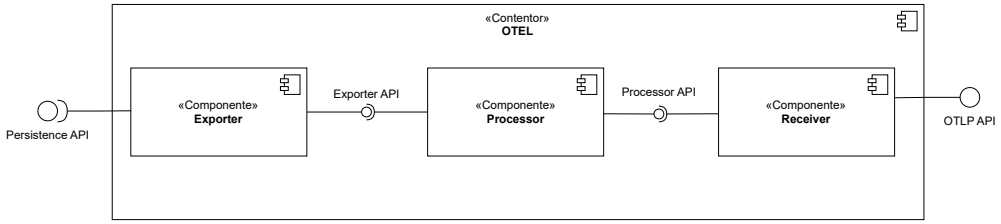

Figura 2.2: OTEL - Desenho Arquitetural.

Como é possível observar, é disponibilizada uma OTLP API que é responsável por receber os dados telemétricos das aplicações monitorizadas, através de um ou mais componentes do tipo *receiver*. Estes mesmos dados telemétricos são depois processados pelo componente

[2]https://prometheus.io/
[3]https://www.jaegertracing.io/

Processor para serem finalmente exportados pelo *Exporter* para a base de dados, nesta caso a ClickHouse.

Para terminar, aquando da escrita deste relatório, o OpenTelemetry é um projeto de código aberto mantido pela comunidade e encontra-se no estado de maturidade *Incubating* da *Cloud Native Computing Foundation* (CNCF). Sendo o segundo mais ativo da mesma, apenas atrás do projeto Kubernetes[4] [20]. Devido às suas características abertas é capaz de evitar o chamado *vendor-lock*. Dito isto, o autor deste relatório pensa que, eventualmente, o projeto OpenTelemetry será, de facto, a solução padrão adotada pelas aplicações nativas em nuvem (do inglês *cloud-native*) capaz de cobrir os três principais pontos da observabilidade - *tracing*, métricas e *logs*.

2.2.2 Prometheus

O Prometheus é uma solução de monitorização cujo código fonte é aberto (do inglês *open-source*). Desde a sua criação pelas mãos de um grupo de desenvolvedores da SoundCloud em 2012, a comunidade abraçou este projeto e um ecossistema tem crescido em volta do mesmo. É um sistema baseado em consultas (do inglês *pull-based system*), ou seja, é o próprio sistema de monitorização que decide quando e que métricas devem ser recolhidas, tendo por base o ficheiro de configurações. É essencialmente desenvolvido com recurso à linguagem de programação *Go* e é licenciado ao abrigo da licença *Apache 2.0* e em 2016, o projeto Prometheus seguiu as pegadas do projeto Kubernetes e tornou-se o segundo membro da CNCF. É capaz de oferecer as ferramentas necessárias para atender as necessidades atuais da Ciberbit, tais como métricas de monitorização e criação de notificações de alertas sobre condições pré-programadas em dispositivos externos. Possui uma estruturação de dados simples mas poderosa e uma linguagem de pesquisa que nos permite analisar o desempenho das nossas aplicações e das nossas infraestruturas [21].

Relativamente à instrumentação das nossas aplicações, existem bibliotecas disponíveis para as mais variadas linguagens de programação, incluindo o Go, o Java e C-Sharp. Soluções como Kubernetes e Docker possuem nativamente integração com estas bibliotecas. Outras soluções de instrumentação podem ser utilizadas, como o OpenTelemetry, e posteriormente ter o Prometheus configurado para comunicar com as mesmas. Ora, o modelo de dados adotado pelo Prometheus regista cada métrica temporal não só pelo nome e pelo *timestamp* correspondente, mas também por uma lista não-ordenada de pares de *key-value* chamados de *labels*. A linguagem de consulta PromQL permite a agregação através de qualquer uma destas *labels*, então é possível não só analisar os dados obtidos por processo, mas também por servidor, por serviço ou por qualquer outra *label* que o utilizador tenha definido. Um formato de dados de texto simples faz com que expor métricas ao Prometheus seja uma tarefa simples [21]. Hoje em dia, múltiplos *softwares*, tanto *Free Open-Soure Software* (FOSS) como comerciais, já possuem suporte para este formato [22].

Dito isto, dentro do Prometheus, é possível também criar alertas usando esta mesma linguagem de consulta - Prometheus *Querying Language* (PromQL). A definição de *labels* faz com que a gestão de alertas seja extremamente fácil nesta plataforma, dado que um único alerta é capaz de cobrir todas as *labels* existentes. Além disso, esta ferramenta possui também a capacidade de descobrir serviços e máquinas que devem ser monitorizados de forma automática de plataformas como o Kubernetes e a Azure [21].

[4] https://kubernetes.io/

Ocasionalmente, as empresas necessitam de monitorizar certos componentes cujas métricas não são passíveis de ser extraídas diretamente. Nomeadamente aquando da execução de processo em lote programados para executar entre intervalos de tempo regulares, por exemplo, a cada hora ou mesmo diariamente. Estes iniciam, realizam algum tipo de processamento e eventualmente terminam. Um vez que eles não estão continuamente a executar, o Prometheus não consegue consultá-los exatamente. Nestes casos, o PushGateway do Prometheus pode ser a solução. O PushGateway permite que os utilizadores enviem métricas personalizadas para um sistema intermediário de onde posteriormente o Prometheus já é capaz de consultas e consumir essas métricas e se necessário criar alertas sob as mesmas. Foi criado com o objetivo de ser utilizado em serviços em lote de curta duração que por iniciativa própria enviam as métricas para este repositório intermédio antes de cessarem operações. No entanto, é de se notar que no caso de, por exemplo, haverem múltiplos envios para o repositório intermediário entre uma consulta do Prometheus e outra, o PushGateway apenas retorna o último envio, os anteriores são descartados. A arquitetura do Prometheus PushGateway é apresentada na Figura 2.3 [21].

Figura 2.3: Arquitetura do Prometheus PushGateway [21]

Dito isto, o Prometheus atende as necessidades e é simples de se incorporar. Um único servidor Prometheus é capaz de digerir milhões de pequenos conjuntos de dados (do inglês *samples*) por segundo. É distribuído como um único ficheiro binário estático - devido ao facto de ser desenvolvido em Go - com apenas um ficheiro de configuração e todos os componentes que formam este serviço podem ser executados a partir de contentores. É desenhado para que possa ser integrado na infraestrutura já existente com o mínimo esforço possível. É importante referir, contudo, que dos três tipos essenciais de telemetria - métricas, *logs* e *distributed traces* - esta solução apenas suporta métricas [21].

2.2.3 Jaeger

Jaeger é uma ferramenta de monitorização de código aberto focada em *distributed traces*. O seu desenvolvimento começou em 2015, pelas mãos de uma equipa de engenheiros da Uber. Foi criado com o objetivo de ajudar a Uber a ter um maior grau de observabilidade dos seus microsserviços de grande escala. Com o passar do tempo, com o aumento da visibilidade deste produto, à semelhança do Prometheus, o mesmo foi entregue à CNCF em 2017 [18]. Juntamente com o Zipkin[5] é considerada uma das ferramentas FOSS mais populares e maduras dentro da área [17].

Como dito anteriormente, diagnosticar problemas em sistemas distribuídos é uma tarefa complexa, dada a natureza destes sistemas. Por exemplo, o simples ato de requisitar uma determinada informação por parte de um utilizador, pode espoletar múltiplos pedidos a

[5] https://zipkin.io/

diferente microsserviços. Estes pedidos podem acontecer concorrentemente e serem independentes entre si. Se nesta requisição, acontecer algum tipo de erro, os desenvolvedores precisam de determinar qual foi microsserviços surgiu o problema [23].

Ora, o Jaeger vem facilitar esta tarefa, ao permitir visualizar as comunicações que acontecem entre os microsserviços. Para isso, à semalhanda do Prometheus, adota nativamente a especificação do OpenTelemetry e disponibiliza uma visualização simples e compacta do modelo de dados da mesma [23]:

- **Span**: Pode ser definido como sendo um bloco lógico de trabalho realizado num sistema distribuído. Por norma, deve ter um nome associado, um tempo de inicio e de fim, algum tipo de etiquetação que facilite o seu agrupamento com outros Spans e estar enquadrado em um contexto específico da sistema (contexto do pedido, por exemplo). Se necessário, pode também ter outros dados associados, como uma descrição, de modo a facilitar o trabalho de depuração dos desenvolvedores;
- **Trace**: Pode ser definido como sendo a coleção de todos os Spans que ocorrem num determinado contexto/pedido.

Esta ferramenta possui uma interface gráfica moderna e intuitiva desenvolvida em React[6]. Segue-se na Figura 2.4, um exemplo da visualização de um *trace* no Jaeger [24]:

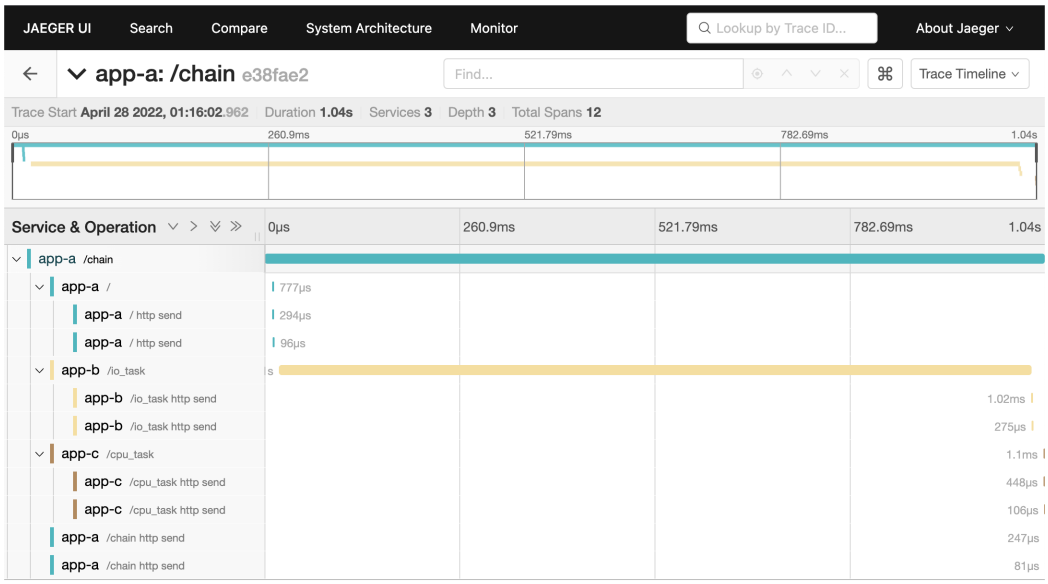

Figura 2.4: Jaeger - Visualização de um Trace [25]

Através dos gráficos de dependências gerados. O Jaeger permite os desenvolvedores acompanharem o pedido pelos diversos serviços no qual passou. É importante referir que o nível de profundidade (do inglês *deep*) pode ser configurado manualmente [24].

Ao contrário do Prometheus, esta ferramenta foi pensada, arquiteturalmente, para ser capaz de receber e processar grandes volumes de dados. A mesma pode ser implantada em múltiplos nós sem qualquer problema, uma vez que foi desenhada para ser capaz de escalar

[6] https://react.dev/

horizontalmente [26]. Por exemplo, na empresa criadora do mesmo (Uber), qualquer instância do Jaeger é responsável tipicamente por processar vários milhares de milhões de *spans* por dia [24].

Contudo, como abordado anteriormente na subsecção 2.1.4, não é recomendado armazenar todos os *traces* registados. Ora, o Jaeger faz uso da técnica *sampling*. Por pré-definição, todas as suas bibliotecas fazem uso de apenas 0.1% de todo o universo de pedidos recebidos, contudo tal valor pode ser alterado manualmente [27].

Tais características fazem com que o Jaeger possa ser considerada uma ferramenta robusta e versátil para a análise de dados telemétricos do tipo *tracing*.

2.2.4 SigNoz

O SigNoz trata-se de uma solução de telemetria completa e apresenta-se como sendo uma alternativa FOSS à popular plataforma de observabilidade DataDog[7] [28]. Ao contrário das duas soluções discutidas anteriormente, este produto é capaz de receber e processar dados telemétricos dos três principais tipos: métricas, *logs* e *traces*. Assim sendo, essa acaba por ser uma das suas principais vantagens, o facto de conseguir reunir a gestão de dados telemétricos em um único lugar.

Ora, o serviço de pesquisa do SigNoz é desenvolvido maioritariamente em Go, uma linguagem de programação conhecida pelo seu desempenho e escalabilidade [29]. Go é uma linguagem compilada o que, aos dias de hoje, ter um melhor desempenho que linguagens de programação como Python ou Ruby [30] [31]. Trata-se também de uma das principais linguagens adotadas em projetos FOSS cujo tema é monitorização, como é o caso do Prometheus[8], do Jaeger[9] ou do próprio OTEL[10]. Por forma a dar uma visão mais profunda ao leitor do serviço em questão é agora apresentada na Figura 2.5 o desenho arquitetural do mesmo:

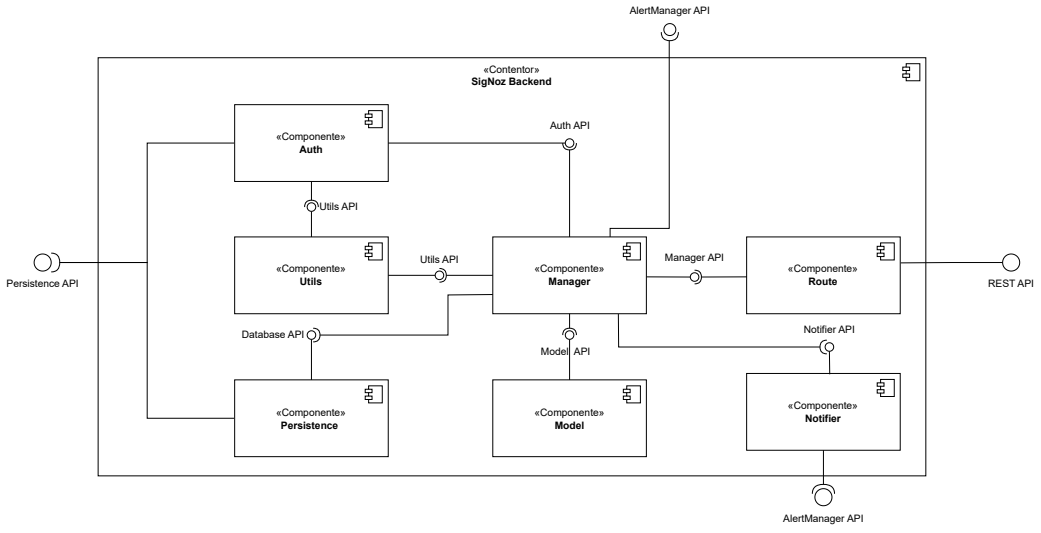

Figura 2.5: SigNoz - Desenho arquitetural da Backend.

[7] https://www.datadoghq.com/
[8] https://github.com/prometheus/prometheus
[9] https://github.com/jaegertracing/jaeger
[10] https://github.com/open-telemetry/opentelemetry-collector

Este contentor é baseado na arquitetura de camadas, com ligeiras alterações à mesma. Por se tratar de uma Backend simples cujo único propósito é criar uma fina camada de abstração entre a Frontend e a base de dados, o componente *Manager* abriga a responsabilidade de toda a camada de negócio da aplicação. Por sua vez, o componente *Persistence* é responsável por comunicar com a base de dados e o componente *Notifier* é encarregado por cuidar do processo de notificações.

Por sua vez, a Frontend é desenvolvida com recurso a TypeScript[11]. Esta linguagem pode ser vista como uma extenção do JavaScript que lhe adiciona tipos. O código produzido em TypeScript é posteriormente traduzido para JavaScript através de um *transpiler* que faz uma análise em busca de possíveis incoerências de tipos. Desta forma, a adoção do TypeScript permite produzir código mais robusto, prevenindo enumeras falhas que possam acontecer em ambientes de produção em JavaScript [32].

No que toca à *framework* escolhida, o SigNoz escolheu usar React[12]. É mantida pela Meta[13] o que permite que tenha uma documentação bastante completa bem como uma comunidade de desenvolvedores ativa de relativa dimensão [33]. Ao contrário do Angular[14] que é uma solução completa para o desenvolvimento de Frontends, o React é uma *framework* bastante minimalista e foca-se somente com a gestão de estados e com a posterior renderização desse mesmo estado para um *Document Object Model* (DOM) virtual, ao fazer uso de um DOM virtual. Dito isto, esta *framework* interliga-se facilmente com outras *frameworks* de modo a completar as suas lacunas. Neste caso específico, foi feito uso da Ant[15] para o desenvolvimento rápido e visualmente agradável da interface gráfica.

Levando em consideração que o SigNoz é uma solução FOSS, a adoção de tecnologias populares permite atrair mais colaboradores e assim permitir que o projeto continue o seu caminho. Neste aspeto, o autor deste relatório considera que a equipa principal de desenvolvimento do SigNoz tomou boas decisões. Dito isto, segue-se na Figura 2.6 o desenho arquitetural da Frontend, por forma a dar uma visão mais profunda ao leitor da Frontend em questão:

[11] https://www.typescriptlang.org/
[12] https://react.dev/
[13] https://about.meta.com/
[14] https://angular.io/
[15] https://ant.design/

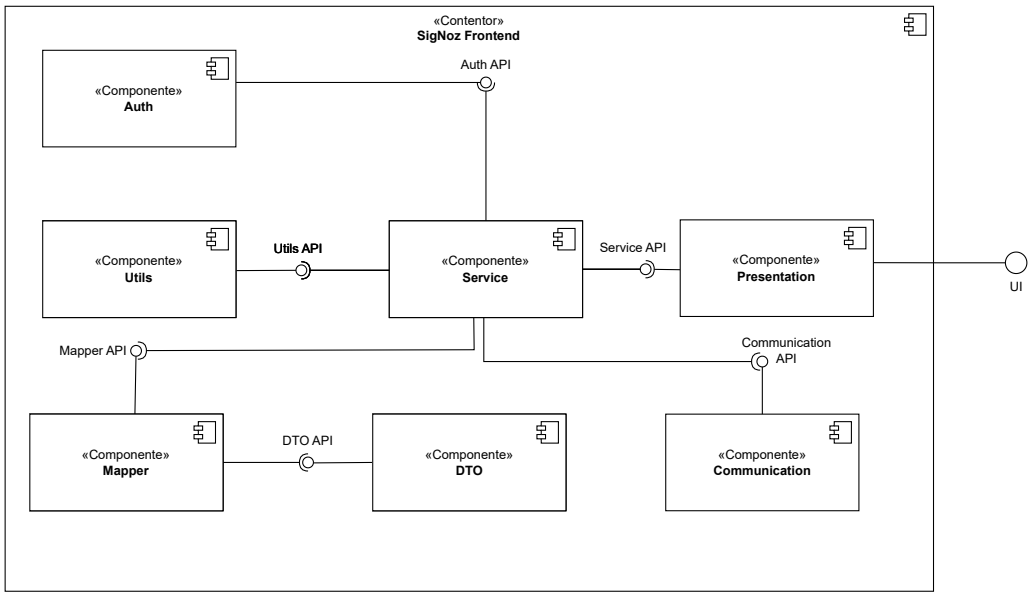

Figura 2.6: SigNoz - Desenho arquitetural da Frontend.

A arquitetura apresentada para a Frontend é baseada na arquitetura de 3 (três) camadas, descrita em maior detalhe na subsecção 4.2.1. De forma sucinta, os componentes representados, o componente *Presentation* diz respeito à camada de apresentação, o componente *Service* faz uso de diversos outros componentes de modo a conseguir realizar as operações necessárias sobre os dados e, portanto, enquadra-se na camada de aplicação. Por fim, o componente *Communication* é responsável por comunicar com o contentor Signoz Backend e diz, por isso, respeito à camada de dados.

Como é posível também observer, este adota também *Single Responsability Principle* (SRP). Para isso, é utilizado o componente Mapper, cujo propósito passa por converter os dados introduzidos pelo utilizador em *Data Transfer Object* (DTO)s. Posteriormente, o componente *Auth* é responsável verificar de antemão quais os recursos é que o utilizador tem acesso e, por último, o componente *Utils* é responsável por disponibilizar diversos métodos que são geralmente utilizados no processamento dos pedidos da Backend.

Por sua vez, no que diz respeito à camada de persistência, é feito uso de uma base dados SQLite[16] para armazenar metadados internos, como a estrutura que suporta a criação dos *dashboards*, as configurações de alertas, os utilizadores da solução, entre outros. Enquanto que este tipo de base de dados é bastante leve e apresenta um bom desempenho mesmo com grandes volumes de dados não é recomendada a sua utilização em cenários onde exista a necessidade de correr múltiplas instâncias da Backend. Isto, porque apesar de a mesma suportar um número ilimitado de serviços a ler em simultâneo, apenas pode haver um a escrever ao mesmo tempo [34]. À data de hoje, não existe suporte oficial por parte da equipa principal de desenvolvedores do SigNoz para outro tipo de base de dados no que toca a este tipo de configurações, apesar de uma integração com a base de dados Postgres[17] - que resolveria o problema das escritas concorrentes - já estar a ser desenvolvida graças ao esforço conjunto da comunidade [35].

[16] https://www.sqlite.org/
[17] https://www.postgresql.org/

Deixando as decisões técnicas de lado, os *dashboards* nativos do SigNoz são bastante intuitivos e ao mesmo tempo poderosos, permitindo efetuar uma série de filtragens e/ou visualizar os mesmos dados de diferentes formas sem grande esforço [36]. Após a execução de uma listagem e/ou filtragem dos *traces* existentes, podemos visualizar as métricas capturadas para esses mesmos *traces*. O Jaeger é incapaz de, nativamente, oferecer uma funcionalidade semelhante a esta. Segue-se na Figura 2.7, um exemplo dessa visualização [36]:

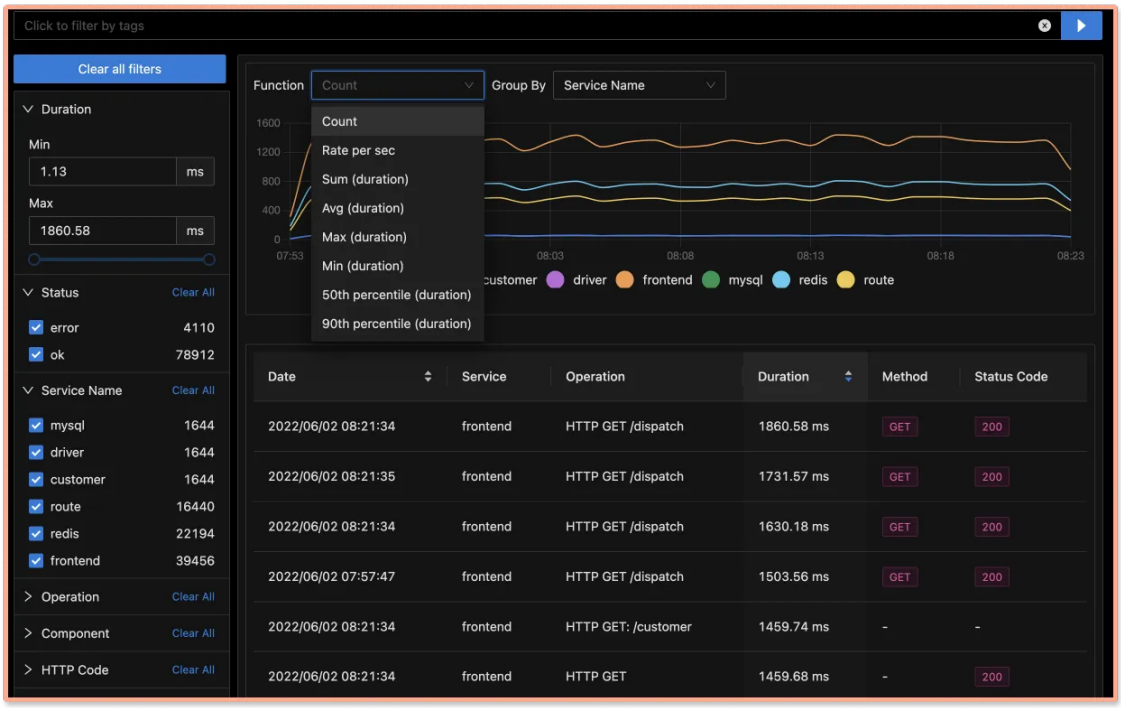

Figura 2.7: SigNoz - Exemplo de filtragem de *traces* [36]

É de realce mencionar que através da instrumentação com recurso às bibliotecas do OpenTelemetry é possível capturar métricas específicas e enquadradas no relevo do modelo de negócio da aplicação. Dito isto, segue-se agora na Figura 2.8, um exemplo de como esta solução apresenta um *trace* específico [36]:

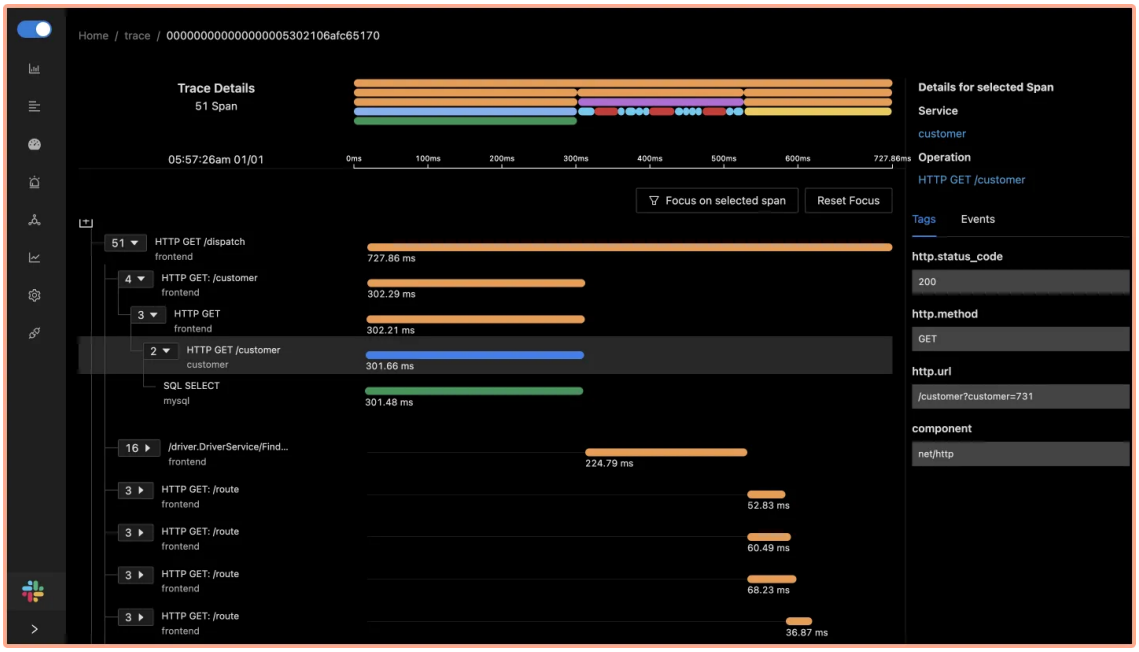

Figura 2.8: SigNoz - Exemplo de visualização de um *trace* [36]

Como é possível observar na imagem, inicialmente na parte superior da página, podemos ter acesso à representação do *trace* através de um *FlameGraph*[18] e mais em baixo, conseguimos visualizar o mesmo *trace* sob a forma de gráfico de Gantt [19]. Fazendo a comparação, o Jaeger não suporta a visualização através de *FlameGraphs* nativamente. Além disso, considerando o contexto do projeto, desde a versão 0.11.3 que é também possível importar *dashboards* através de estruturas *JavaScript Object Notation* (JSON) da ferramenta Grafana[20] para uma melhor visualização dos dados telemétricos capturados. Esta ferramenta é capaz de produzir gráficos de análise bastantes completos que posteriormente o SigNoz é capaz de ingerir e preencher com dados telemétricos [36].

Além disso, o SigNoz permite também, de forma nativa, criar alertas com base na PromQL anteriormente mencionada na ferramenta Prometheus [36]. Alertas esses que depois se integram sem esforço com a interface gráfica, como é possível observar na Figura 2.9:

[18] https://www.datadoghq.com/knowledge-center/distributed-tracing/flame-graph/
[19] https://www.gantt.com/
[20] https://grafana.com/

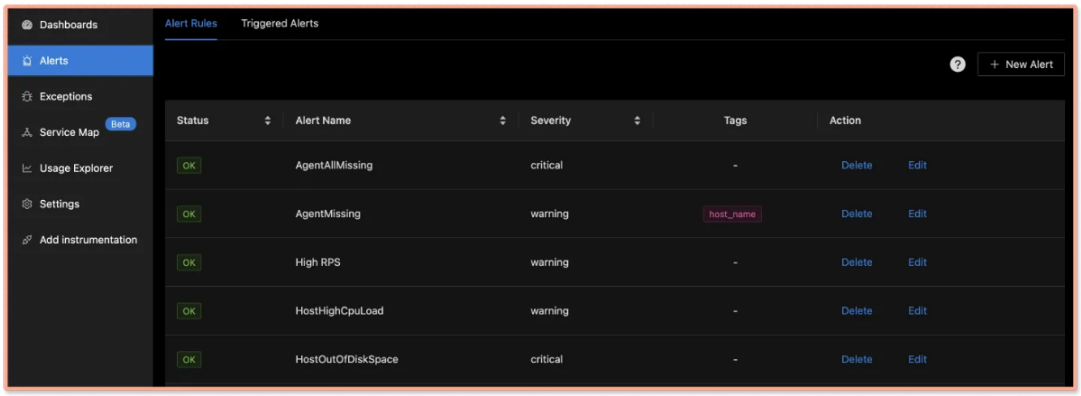

Figura 2.9: SigNoz - Exemplo de listagem de regras de alertas. [36]

Concluindo, tratando-se o SigNoz de uma solução de observabilidade completa, o mesmo é capaz de satisfazer as necessidades da empresa proponente, quer seja em termos de métricas ou de *traces*.

2.2.5 Sumário

Em suma, nesta subsecção abordaram-se 4 (quatro) soluções existentes. Onde 3 (três) são soluções de análise passíveis de serem utilizadas em conjunto com o projeto OpenTelemetry e assim satisfazer os objetivos propostos. Começou-se por analisar uma ferramenta focada apenas em métricas, posteriormente uma focada somente em *traces* e por fim, uma solução de observabilidade completa. Através de uma pesquisa cuidada das características destas três soluções e levando em consideração o facto da Ciberbit ter preferência por uma plataforma única, optou-se por se adotar o SigNoz. A esta razão, acresce o facto de que durante as breves experiências que o autor realizou nas 3 (três) ferramentas, esta foi também aquela que o autor considerou mais confortável em usar.

É de referir que devido ao facto da especificação da OpenTelemetry relativamente a *logs* ainda não se encontrar estável aquando da escrita deste relatório [37], bem como este não ser crucial para o cumprimento dos objetivos iniciais, nenhuma solução com foco específico neste pilar da telemetria foi analisada. É também importante mencionar que apesar disso, a solução selecionada - SigNoz - encontra-se, à data, devidamente preparada para receber *logs* formatadas através da versão mais recente da especificação experimental da OpenTelemetry [38].

Capítulo 3

Análise

Neste capítulo é feita uma análise do problema existente, onde inicialmente é apresentada uma breve análise da solução atualmente em vigor na empresa proponente e posteriormente é realizado um levantamento dos requisitos que a solução final procura cumprir.

3.1 Análise da solução

Esta secção tem como objetivo apresentar o problema em questão tendo por base a análise da solução existente, do que esta é capaz de realizar e as suas limitações que a impedem de satisfazer as necessidades atuais da empresa proponente.

Como referido anteriormente neste relatório, a solução existente é dotada de mecanismos capazes de capturar uma variedade considerável de dados telemétricos, métricas e *logs*, do I'MTHOM através de uma REST API. Ora, quando acedemos à mesma, designada internamente e doravante nesta secção por Status Monitor, é apresentada uma página de autenticação. Este mecanismo permite averiguar a autenticidade do utilizar, bem como o nível de acesso que o mesmo possui e consequentemente adaptar a interface gráfica ao mesmo. Nem todos os utilizadores têm acesso ao mesmo conjunto de informações. A figura 3.5 ilustra a página de autenticação. É de se notar que atualmente não existe nenhum mecanismo de registo implementado. Para se adicionar um novo utilizador, é necessário fazê-lo através da base de dados.

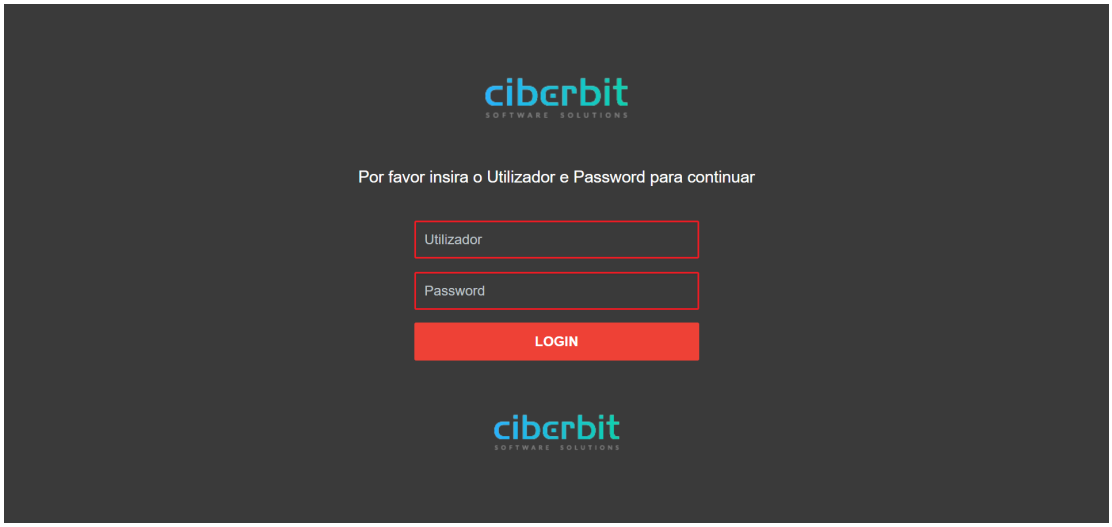

Figura 3.1: Status Monitor - Mecanismo de autenticação.

Realizada a autenticação, é apresentado uma conjunto de opções que tem por base o nível de acesso dos grupos aos quais o utilizador participa. De modo a se conseguir apresentar as principais funcionalidades do Status Monitor, entrou-se com uma conta do grupo Administradores. Para este grupo, a figura 3.6 mostra o menu apresentado a colaboradores com este nível de acesso.

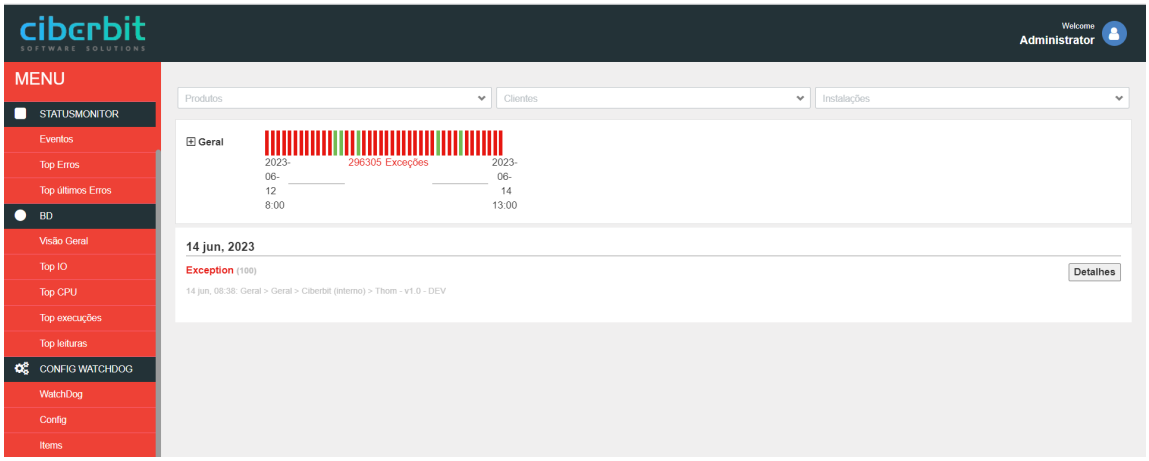

Figura 3.2: Status Monitor - Menu Inicial.

A página inicial mostra o número de exceções que aconteceram na instância de desenvolvimento do I'MTHOM nos últimos 3 (três) dias. No contexto do Status Monitor, todos os registos telemétricos são considerados eventos, ou seja, uma exceção é um tipo de evento. Na parte inferior da página é também apresentada uma lista com as 100 exceções mais recentes capturadas. A esta lista é possível aplicar filtros. Atualmente é possível filtrar por produto, cliente e instalação.

Do lado esquerdo da imagem, o menu lateral permite ter acesso a outras informações mais pormenorizadas. O primeiro grupo diz respeito aos eventos capturados pelo Status Monitor. Segue-se na figura 3.7 uma lista ilustrativa dos eventos reportados pelo I'MTHOM ao Status Monitor, obtida a partir da primeira opção deste grupo. Estes eventos podem-se tratar de exceções, operações mal-sucedidas, latências exageradas nas comunicações com respetivos tempos, entre outros.

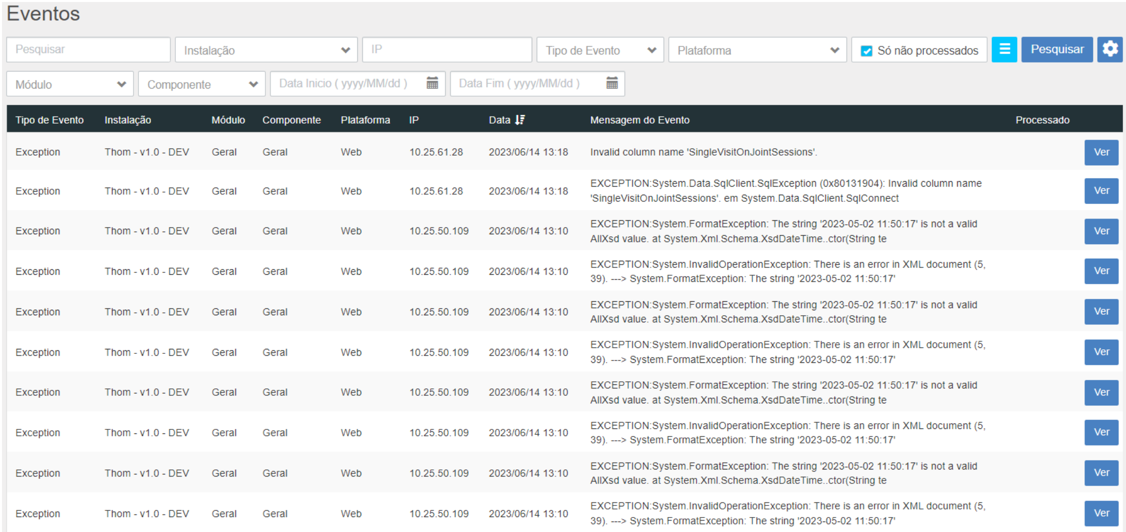

Figura 3.3: Status Monitor - Lista de eventos.

À semelhança do que acontecia no menu anterior, é possível aplicar um conjunto de filtros a estes eventos e posteriormente exportar o resultado dos mesmos para um ficheiro *Comma-separated values* (CSV). Esta plataforma possui, contudo, uma limitação nesta e em outras filtragens que se traduz em tempos de resposta demasiado longos e, em certos casos, *timeout*. A esta, acresce o facto de as outras duas opções deste grupo ainda não estarem implementadas. O objetivo é que a segunda opção - Top Erros - mostre os erros mais frequentes e que a terceira opção - Top últimos Erros - mostre os últimos erros capturados. Determina-se que neste contexto, um erro trata-se de uma exceção e/ou operação mal sucedida.

Dito isto, o segundo grupo visível também não possui qualquer tipo de implementação. No entanto, a ideia passa por mostrar diversas métricas referentes às múltiplas bases de dados em uso pelos produtos monitorizados, como listas de consultas que mais consomem CPU, mais tempo demoram, mais vezes são executadas, entre outras.

No que diz respeito ao terceiro grupo, grupo relativo aos diferentes WatchDogs, é possível na primeira secção consultar a lista de todas as instâncias deste tipo de serviço. Na segunda secção é possível visualizar a configuração de cada um. Cada WatchDog pode ter várias configurações associadas com diversas funções integradas para monitorizar diversos *endpoints*. Cada configuração está, posteriormente, associada a uma instância de um produto, como por exemplo, à instância de desenvolvimento do I'MTHOM. Através da interface gráfica é possível editar ou eliminar os elementos presentes, sendo que todas estas alterações são posteriormente persistidas na base de dados.

Ora, na prática, o WatchDog vai recolher a sua configuração a um *endpoint* disponibilizado pelo Status Monitor e posteriormente através dessa configuração vai monitorizar um conjunto de aplicações/instâncias enviar os dados recolhidos de volta para o Status Monitor. Trata-se assim de um mecanismo ativo de monitorização em contraste com o mecanismo comum onde é a aplicação monitorizada quem envia os dados telemétricos. As imagens ilustrativas encontram-se presentes no Apêndice A.

Em suma, como já mencionado no capítulo 1, a solução atual não é capaz de apresentar dados telemétricos suficientes para oferecer um bom grau de observabilidade aos utilizadores finais. Esta limitação é, contudo, na sua maioria responsabilidade da falta de instrumentação dos próprios produtos, neste caso do próprio I'MTHOM. Para além disso, não existe qualquer processamento ou análise com intuito de tirar valor dos dados capturados, a isto acresce o facto não existirem estruturas visuais criadas que suportem uma mais fácil visualização em prol das já apresentadas listagens.

3.2 Engenharia de Requisitos

A área de engenharia de requisitos permite identificar e analisar as necessidades e restrições da solução a desenvolver, sendo, por isso, um importante artefacto de análise, para a implementação de qualquer produto de *software*. Ora, podemos definir um requisito como sendo uma condição ou capacidade possuída pelo *software* de maneira a resolver um problema do mundo real. Estes problemas podem, por exemplo, estar relacionados com a necessidade de automatizar partes de um sistema, corrigir deficiências ou implementar novas funcionalidades [39]. Dito isto, esta secção visa apresentar os requisitos funcionais e não funcionais, também designados por atributos de qualidade, sob os quais a solução deve assentar. Este atributos incluem preocupações como restrições tecnológicas, tempos de resposta, entre outros fatores que influenciem o desenho e consequentemente a implementação e implantação da *pipeline*.

3.2.1 Requisitos funcionais

Os requisitos funcionais dizem respeito às funcionalidades que a solução final deve possuir. Estas encontram-se nesta subsecção representadas sob a forma de casos de uso e posteriormente explicadas. Os casos de uso representam ações passíveis de serem realizadas por um ou mais atores do sistema com o intuito de atingir um determinado objetivo. Ora, uma forma de apresentar os mesmos de forma clara e coesa é através da realização de um diagrama de casos de uso, que pode ser observado na Figura 3.4.

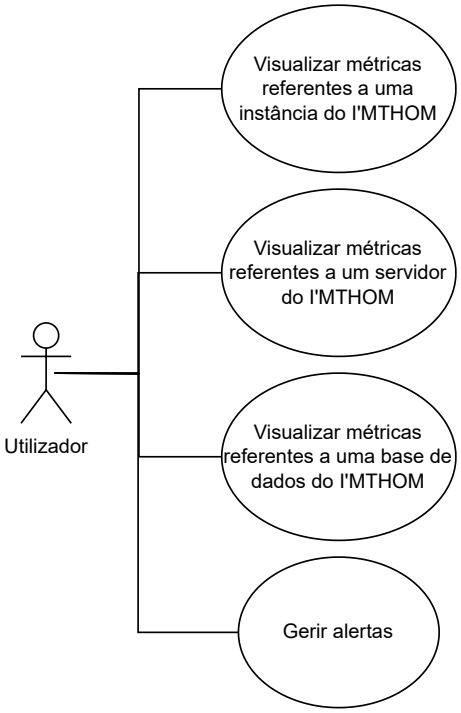

Figura 3.4: Diagrama de Casos de Uso

Como pode-se observar, a camada de visualização da *pipeline*, por agora, pressupõe a existência de apenas um papel de utilizador genérico. Dito isto, segue-se uma enumeração mais detalhada dos casos de uso apresentados:

- Visualizar métricas referentes a uma instância: dados estatísticos sobre uma determinada instância do I'MTHOM como, por exemplo, o número de utilizadores conectados, a latência observada aquando dos pedidos ao longo de tempo, entre outros;
- Visualizar métricas referentes a um servidor: dados estatísticos sobre um determinado servidor como, por exemplo, o uso do CPU ao longo de um período de tempo, o mesmo para a memória principal entre outros recursos;
- Visualizar métricas referentes a uma base de dados: dados estatísticos relativos à base de dados como, por exemplo, as consultas que mais consomem CPU, as consultas que mais tempo demoram, entre outros;
- Gerir alertas: criação e edição de regras de alertas. Estes alertas devem ser enviados através do uso de Webhook ou email.

É importante referir que através da camada de visualização da *pipeline* deve ser possível executar filtros pelas métricas capturadas e enunciadas nos casos de uso supracitados. Dito isto, tendo por base o último caso de uso enumerado, segue-se agora um levantamento dos principais alertas que a empresa proponente tenciona ver criados, tendo por base as métricas enumeradas na secção 1.3:

- Alto consumo de CPU: Quando o uso de CPU por parte do servidor de uma instância do I'MTHOM ou da base de dados correspondente ultrapassar um determinado limiar num

ponto específico do tempo ou durante um intervalo de tempo, o que pode significar que a aplicação não esteja a realizar as suas operações da forma mais eficiente possível;

- Alto consumo de memória RAM: Quando o uso de memória RAM por parte do servidor de uma instância do I'MTHOM ou da base de dados correspondente ultrapassar um determinado limiar num ponto específico do tempo ou durante um intervalo de tempo, o que pode indicar uma má gestão da memória em código;
- Latência de pedidos: Quando a latência de um pedido é superior a um determinado limiar, o que pode significar que a infraestrutura de rede pode ter problemas de desempenho;
- Volume de transações anormal: Quando o número de transações num determinado espaço de tempo excede um determinado limiar, o que pode significar que o sistema se encontra sob uso intenso;

Em conjunto, estes alertas permitem que a Ciberbit identifique problemas que possam existir com o seu principal produto bem como a base de dados que sustenta o mesmo num espaço de tempo mais curto do que aquele que acontece atualmente. Ao ser capaz de responder a estas situações em tempo útil através dos alertas espoletados, a empresa proponente mais facilmente será capaz de fornecer serviços fiáveis e de alto desempenho às unidades de saúde que serve.

3.2.2 Atributos de qualidade

A complexidade de um produto de *software* é determinada em parte pelas suas funcionalidades, ou seja, o que o produto consegue realizar e também pelos atributos globais durante a fase de desenvolvimento, por exemplo, custos operacionais, desempenho, fiabilidade, manutenibilidade, portabilidade, robustez, entre outros. Estes atributos de qualidade, também conhecidos como atributos não funcionais, têm um papel crítico durante a fase de desenvolvimento, uma vez que servem como critérios de seleção de arquiteturas e tecnologias [40].

No âmbito do projeto em causa, o processo de análise e levantamento de atributos não funcionais teve em consideração as características das soluções estudadas na secção 2.2, bem como dissertações e relatórios técnicos que abordam a mesma temática. Em seguida encontram-se listados os mesmos:

- R.NF.1: A camada de instrumentação da *pipeline* deve de afetar o mínimo possível as aplicações já existentes;
- R.NF.2: A *pipeline* deve ser projetada de forma a facilitar alterações à mesma;
- R.NF.3: A camada de visualização da *pipeline* apenas deve permitir o acesso aos dados telemétricos a utilizadores autenticados;
- R.NF.4: A camada de visualização da *pipeline* deve realizar todas as comunicações com o utilizador final através de protocolos considerados seguros (p.ex. *Hypertext Transfer Protocol Secure* (HTTPS);
- R.NF.5: A camada de visualização *pipeline* deve disponibilizar um meio para analisar e visualizar a informação através de listagens, bem como gráficos correspontes;
- R.NF.6: A *pipeline* deve ser capaz de notificar sobre a ocorrência de condições pré-configuradas ou anomalias aos utilizadores através do uso de Webhooks ou email;

- R.NF.7: A *pipeline* deve de garantir a integridade dos dados telemétricos quando estes são transportados;
- R.NF.8: A *pipeline* deve procurar garantir a autenticidade dos dados telemétricos quando estes são transportados;
- R.NF.9: A camada de visualização da *pipeline* deve suportar os principais navegadores existentes (Chrome, Safari, Edge, Firefox, Opera);
- R.NF.10: A *pipeline* deve permitir uma escalabilidade horizontal;
- R.NF.11: A camada de processamento da *pipeline* deve expor uma API;
- R.NF.12: A camada de visualização da *pipeline* deve suportar múltiplas sessões em simultâneo;
- R.NF.13: A *pipeline* deve ser capaz de comunicar com o exterior utilizando os protocolos HTTP e *Simple Mail Transfer Protocol* (SMTP);
- R.NF.14: A *pipeline* deve ser capaz de, em 90% dos casos, quando uma regra de alarme se verificar, notificar os utilizadores em 15 segundos.
- R.NF.15: As camadas de processamento e de visualização da *pipeline* devem ser capazes de recuperar de um estado de falha.

Tendo por base os atributos supracitados o próximo passo consiste em categorizá-los. Com isto em mente, optou-se por adotar o modelo FURPS+. Este modelo é utilizado de forma a validar os atributos mais priorizados pelas necessidades do cliente. O acrónimo representa categorias que são utilizadas na definição dos atributos de qualidade [41]. Segue-se, então, a categorização dos atributos:

- F (Funcionalidade): esta categoria especifica as funcionalidades que não se relacionam com os casos de uso, como auditoria, interoperabilidade e segurança [42]. Dito isto, no contexto deste projeto foram identificados os atributos R.NF.3 e R.NF.6 como atributos de funcionalidade;
- U (Usabilidade): esta categoria avalia a interface com o utilizador - prevenção de erros, estética e design, ajudas, documentação, consistência e padrões [42]. Dito isto, no contexto deste projeto foi identificado o requisito R.NF.5 como requisito de usabilidade;
- R (Confiabilidade): esta categoria refere-se à integridade, conformidade e interoperabilidade-frequência e gravidade de falhas, possibilidade de recuperação, extensão e duração da falha (valorização/sobrevivência) e previsibilidade (estabilidade) [42]. Dito isto, no contexto deste projeto foram identificados os atributos R.NF.7, R.NF.8 e R.NF.15 como atributos de confiabilidade;
- P (Desempenho): esta categoria avalia os atributos de desempenho de *software* - tempo de resposta, consumo de recursos (energia, CPU, memória, etc.) [42]. Dito isto, no contexto deste projeto foi identificados os atributos R.NF.12 e R.NF.14 como atributos de desempenho;
- S (Suportabilidade): esta categoria refere-se à testabilidade, adaptabilidade, manutenibilidade, compatibilidade, configurabilidade, escalabilidade, entre outros [42]. Dito isto, no contexto deste projeto foram identificados os atributos R.NF.1, R.NF.2, R.NF.9 e R.NF.10 como atributos de suportabilidade.

Por último, o símbolo + do acrónimo FURPS+ abrange restrições que não se enquadram nas categorias previamente mencionadas e que, por isso, se encontram de seguida enunciadas:

- Restrições de desenho: restrições que limitem o desenho do sistema - p.ex. tipo de bases de dados [41]. Dito isto, no contexto deste projeto não foi identificado qualquer requisito como sendo uma restrição de desenho;
- Restrições de implementação: restrições que obriguem o uso de uma tecnologia em específico - p.ex. linguagem de programação, biblioteca ou padrões de construção específicos [41]. Dito isto, no contexto deste projeto não foi identificado qualquer requisito como sendo uma restrição de implementação;
- Restrições de interface: restrições que indiquem os mecanismos de interação obrigatórios com o exterior [41]. Dito isto, no contexto deste projeto foram identificados os atributos R.NF.4, R.NF.11 e R.NF.13 como sendo uma restrição de interface;
- Restrições físicas: restrições físicas relativas ao *hardware* do sistema - p.ex. tamanho e peso [41]. Dito isto, no contexto deste projeto não foi identificado qualquer requisito como sendo uma restrição física.

Esta secção tem como objetivo apresentar o problema em questão tendo por base a análise da solução existente, do que esta é capaz de realizar e as suas limitações que a impedem de satisfazer as necessidades atuais da empresa proponente.

Como referido anteriormente neste relatório, a solução existente é dotada de mecanismos capazes de capturar uma variedade considerável de dados telemétricos, métricas e *logs*, do I'MTHOM através de uma REST API. Ora, quando acedemos à mesma, designada internamente e doravante nesta secção por Status Monitor, é apresentada uma página de autenticação. Este mecanismo permite averiguar a autenticidade do utilizar, bem como o nível de acesso que o mesmo possui e consequentemente adaptar a interface gráfica ao mesmo. Nem todos os utilizadores têm acesso ao mesmo conjunto de informações. A figura 3.5 ilustra a página de autenticação. É de se notar que atualmente não existe nenhum mecanismo de registo implementado. Para se adicionar um novo utilizador, é necessário fazê-lo através da base de dados.

Figura 3.5: Status Monitor - Mecanismo de autenticação.

Realizada a autenticação, é apresentado uma conjunto de opções que tem por base o nível de acesso dos grupos aos quais o utilizador participa. De modo a se conseguir apresentar as principais funcionalidades do Status Monitor, entrou-se com uma conta do grupo Administradores. Para este grupo, a figura 3.6 mostra o menu apresentado a colaboradores com este nível de acesso.

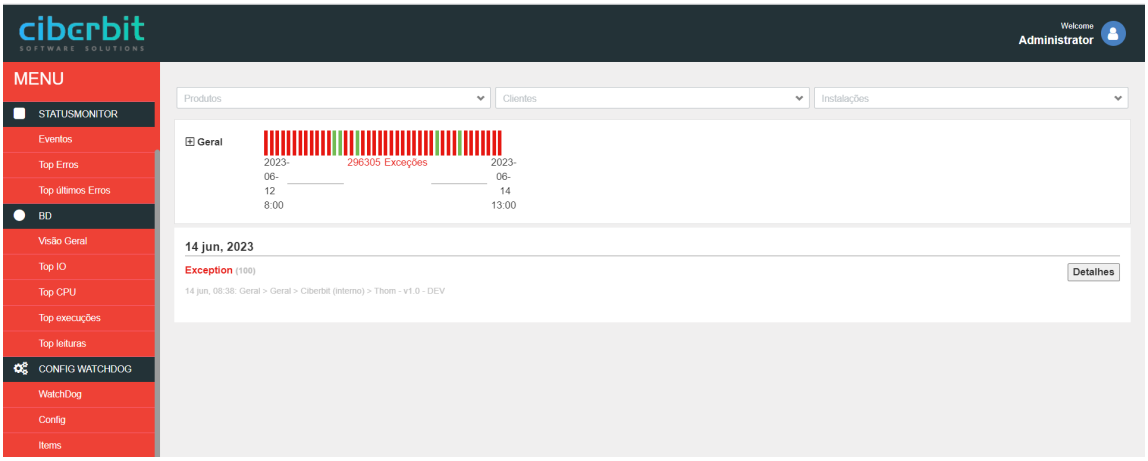

Figura 3.6: Status Monitor - Menu Inicial.

A página inicial mostra o número de exceções que aconteceram na instância de desenvolvimento do I'MTHOM nos últimos 3 (três) dias. No contexto do Status Monitor, todos os registos telemétricos são considerados eventos, ou seja, uma exceção é um tipo de evento. Na parte inferior da página é também apresentada uma lista com as 100 exceções mais recentes capturadas. A esta lista é possível aplicar filtros. Atualmente é possível filtrar por produto, cliente e instalação.

Do lado esquerdo da imagem, o menu lateral permite ter acesso a outras informações mais pormenorizadas. O primeiro grupo diz respeito aos eventos capturados pelo Status Monitor. Segue-se na figura 3.7 uma lista ilustrativa dos eventos reportados pelo I'MTHOM ao Status Monitor, obtida a partir da primeira opção deste grupo. Estes eventos podem-se tratar de exceções, operações mal-sucedidas, latências exageradas nas comunicações com respetivos tempos, entre outros.

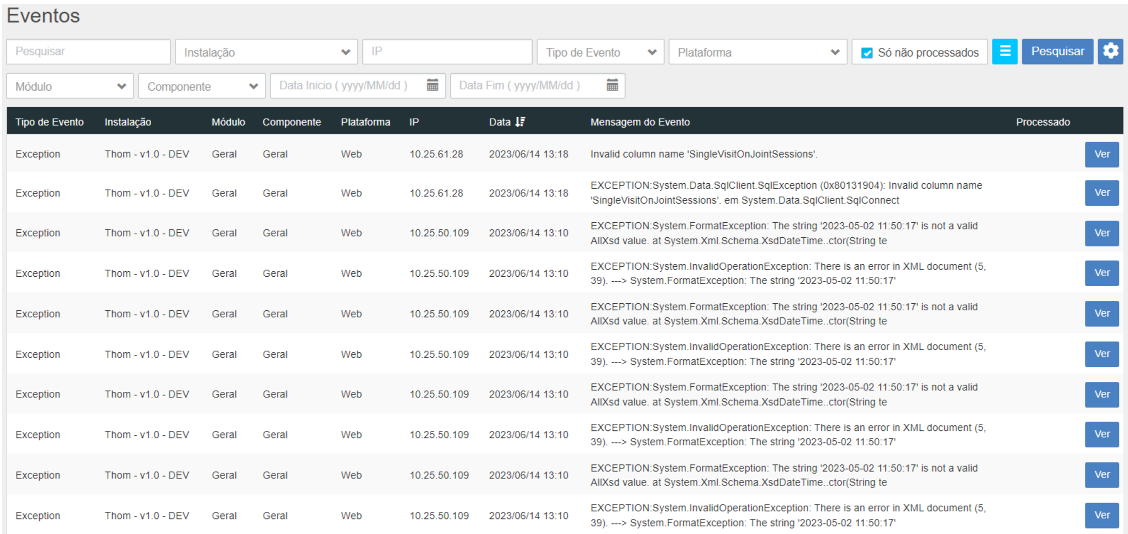

Figura 3.7: Status Monitor - Lista de eventos.

À semelhança do que acontecia no menu anterior, é possível aplicar um conjunto de filtros a estes eventos e posteriormente exportar o resultado dos mesmos para um ficheiro CSV. Esta plataforma possui, contudo, uma limitação nesta e em outras filtragens que se traduz em tempos de resposta demasiado longos e, em certos casos, *timeout*. A esta, acresce o facto de as outras duas opções deste grupo ainda não estarem implementadas. O objetivo é que a segunda opção - Top Erros - mostre os erros mais frequentes e que a terceira opção - Top últimos Erros - mostre os últimos erros capturados. Determina-se que neste contexto, um erro trata-se de uma exceção e/ou operação mal sucedida.

Dito isto, o segundo grupo visível também não possui qualquer tipo de implementação. No entanto, a ideia passa por mostrar diversas métricas referentes às múltiplas bases de dados em uso pelos produtos monitorizados, como listas de consultas que mais consomem CPU, mais tempo demoram, mais vezes são executadas, entre outras.

No que diz respeito ao terceiro grupo, grupo relativo aos diferentes WatchDogs, é possível na primeira secção consultar a lista de todas as instâncias deste tipo de serviço. Na segunda secção é possível visualizar a configuração de cada um. Cada WatchDog pode ter várias configurações associadas com diversas funções integradas para monitorizar diversos *endpoints*. Cada configuração está, posteriormente, associada a uma instância de um produto, como por exemplo, à instância de desenvolvimento do I'MTHOM. Através da interface gráfica é possível editar ou eliminar os elementos presentes, sendo que todas estas alterações são posteriormente persistidas na base de dados.

Ora, na prática, o WatchDog vai recolher a sua configuração a um *endpoint* disponibilizado pelo Status Monitor e posteriormente através dessa configuração vai monitorizar um conjunto de aplicações/instâncias enviar os dados recolhidos de volta para o Status Monitor. Trata-se assim de um mecanismo ativo de monitorização em contraste com o mecanismo comum onde é a aplicação monitorizada quem envia os dados telemétricos. As imagens ilustrativas encontram-se presentes no Apêndice A.

Em suma, como já mencionado no capítulo 1, a solução atual não é capaz de apresentar dados telemétricos suficientes para oferecer um bom grau de observabilidade aos utilizadores

finais. Esta limitação é, contudo, na sua maioria responsabilidade da falta de instrumentação dos próprios produtos, neste caso do próprio I'MTHOM. Para além disso, não existe qualquer processamento ou análise com intuito de tirar valor dos dados capturados, a isto acresce o facto não existirem estruturas visuais criadas que suportem uma mais fácil visualização em prol das já apresentadas listagens.

Capítulo 4

Desenho

Como referido na secção 2.2, uma *pipeline* de observabilidade consiste essencialmente em 3 (três) camadas. Por forma a que todo este processo seja eficiente, é preciso que o mesmo seja desenhado segundo alguns princípios base. Por exemplo, o código responsável por capturar os dados telemétricos deve tentar ter a execução mais leve possível em termos de recursos, de modo a não afetar o desempenho do I'MTHOM. Relativamente à segunda etapa, os dados telemétricos devem ser capturados e armazenados de uma forma fiável e escalável, de modo a análise que aconteça posteriormente seja o mais rápida e precisa possível. Esta análise, por sua vez, deve ser realizada em tempo real ou muito próximo disso de modo a que os possíveis erros que aconteçam durante a execução do I'MTHOM sejam o mais rapidamente tratados. Por fim, a interface gráfica com os dados telemétricos deve procurar ser o mais intuitiva e fácil de navegar possível [43].

Dito nisto, neste capítulo, é apresentada uma descrição de toda a *pipeline*, com foco no seu nível arquitetural e para tal, é utilizada uma combinação de dois modelos de representação arquiteturais, o C4 e o 4+1. Desta forma pretende-se que o leitor obtenha uma compreensão mais abrangente e clara do funcionamento do sistema em questão.

O modelo arquitetural C4 tem como finalidade descrever a arquitetura de um produto de *software* através de quatro níveis de abstração, com cada nível apresentando uma granularidade mais fina que o anterior, o que permite a visualização de um maior conjunto de detalhes de uma parte específica do sistema. Como este modelo apresenta um conjunto bastante limitado de níveis, a sua utilização acaba por ser prática e simples. Importa também salientar que não é obrigatório representar todos os níveis, mas sim aqueles que contribuem para enriquecer o projeto [44].

Segue-se uma breve descrição dos níveis supracitados [44]:

- **Nível 1**: Apresenta os diagramas de contexto, é aquele que apresenta um maior nível de abstração, explicando o sistema como um todo. Pode também incorporar outros sistemas que comuniquem com o sistema principal da solução;
- **Nível 2**: Apresenta as várias partes do sistema, denominadas contentores, que podem se referir a aplicações ou servidores utilizados para o armazenamento de dados, tem como principal objetivo mostrar as escolhas tecnológicas realizadas bem como de que forma é os contentores se comunicam entre si;
- **Nível 3**: Apresenta o diagrama de componentes, isto é, amplia os contentores do nível anterior e apresenta os múltiplos componentes que fazem parte dos mesmos, apresentando para cada um as suas responsabilidades;

- **Nível 4**: Por fim, este nível apresenta como cada componente do nível anterior pode ser implementado em código. Este nível é, contudo, considerado opcional.

Por sua vez, o modelo de vistas 4+1 tem como principal objetivo proporcionar uma visão ampla e completa do sistema através de um conjunto de diversas vistas complementares entre si que, em conjunto, permitem a análise de diversos requisitos dos vários *stakeholders* da solução, tais como os próprios utilizadores, programadores, entre outros, de maneira separada e detalhada [45].

Seguem-se as 5 (cinco) vistas deste modelo [45]:

- **Vista lógica**: apresenta os múltiplos aspetos do *software*. Visa mostrar os conceitos de negócio e as partes lógicas do sistema;
- **Vista de processos**: apresenta, de forma simplificada, o fluxo de funcionamento entre as diversas partes que constituem o sistema através de interações, por forma a ilustrar a realização dos processos de negócio;
- **Vista de implementação**: apresenta como o código produzido se encontra organizado;
- **Vista física**: apresenta os diversos componentes físicos do sistema;
- **Vista de cenários**: apresenta a relação entre os autores e os cenários do sistema.

Ora, a utilização conjunta destes dois modelos permite representar o sistema a partir de múltiplas perspetivas, cada uma com seu próprio nível de detalhe. Podemos, por exemplo, representar uma vista específica do modelo 4+1 com vários graus de granularidade, utilizando o modelo C4.

Feita uma breve introdução teórica sobre ambos os modelos adotados, neste capítulo são apresentados os diagramas de vista lógica e vista física produzidos com recurso à UML. Optou-se por não produzir as restantes vistas do modelo 4+1 por não se considerar relevantes para este projeto em específico, bem como os níveis 3 e 4 do modelo C4. O nível 3, apesar de na opinião do autor, o detalhe que este exige não se adequar ao projeto em questão, é de relevo mencionar que alguns exemplos de diagramas arquiteturais de nível 3 foram já apresentados nas secções 2.2.1 e 2.2.4, por forma a agregar detalhes importantes ao leitor das ferramentas selecionadas. Por fim, não foi produzido o nível 4, por este já não corresponder a desenho arquitetural.

4.1 Nível 1

Nesta secção é apresentada a vista lógica de nível 1 (um). Esta vista é capaz de, a partir de um nível de granularidade elevado, expor a *pipeline* de observabilidade como um todo. É de realce mencionar que não é representada a vista física, por se considerar que com o grau de abstração exigido por este nível, não se iria mostrar qualquer informação relevante ao leitor.

Dito isto, segue-se na Figura 4.1 a vista lógica do sistema:

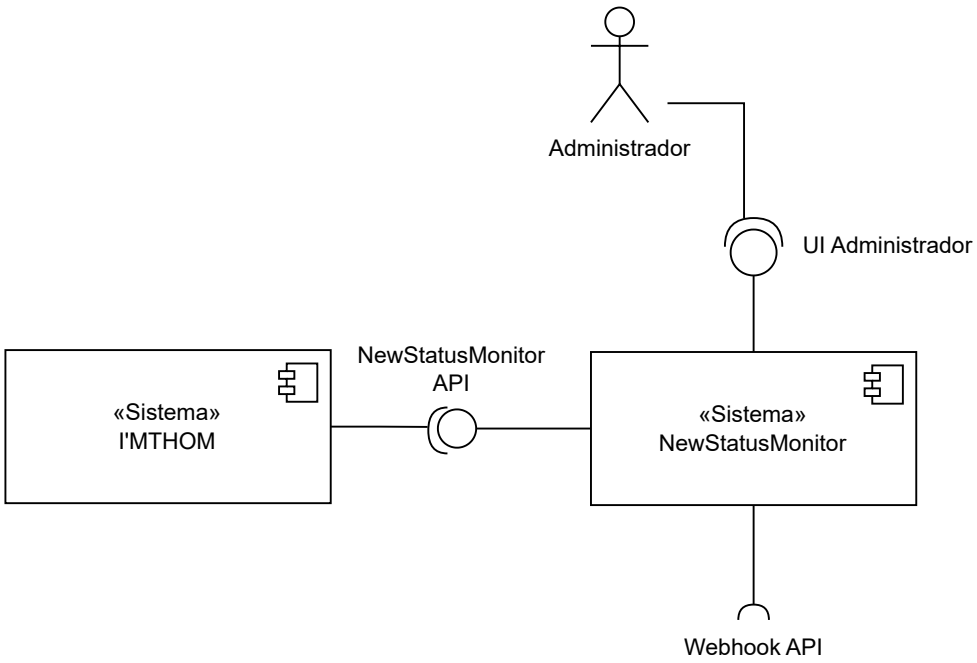

Figura 4.1: Vista Lógica de Nível 1.

Como se pode observar, o sistema NewStatusMonitor disponibiliza uma API por forma ser a capaz de receber dados do exterior. Esta API tem como principal objetivo receber dados telemétricos dos sistemas monitorizados. Posteriormente, recai sobre este a responsabilidade de processar e armazenar de forma eficiente os dados recebidos e apresentá-los posteriormente de forma visualmente intuitiva. Como já referido anteriormente, no contexto do problema, o principal sistema externo que providenciará dados telemétricos é o I'MTHOM, representado neste diagrama e nos futuros como uma caixa negra.

É de realce referir que apesar de atualmente apenas se considerar a existência de um tipo de utilizador (Administrador), o sistema SigNoz deve estar configurado para ser capaz de disponibilizar diferentes interfaces gráficas para diferentes tipos de utilizadores, ou mais concretamente, para utilizadores com diferentes níveis de acesso. Este sistema deve também de ser capaz de enviar as notificações relativas aos alertas que se encontrem ativos.

4.2 Nível 2

Nesta secção são explorados os diversos contentores que formam o sistema. Estes contentores, por sua vez, apresentam um maior nível maior de detalhe do que aquele que foi apresentado na secção anterior.

4.2.1 Vista Lógica

Segue-se na Figura 4.2, a vista de lógica de nível 2 (dois) da solução:

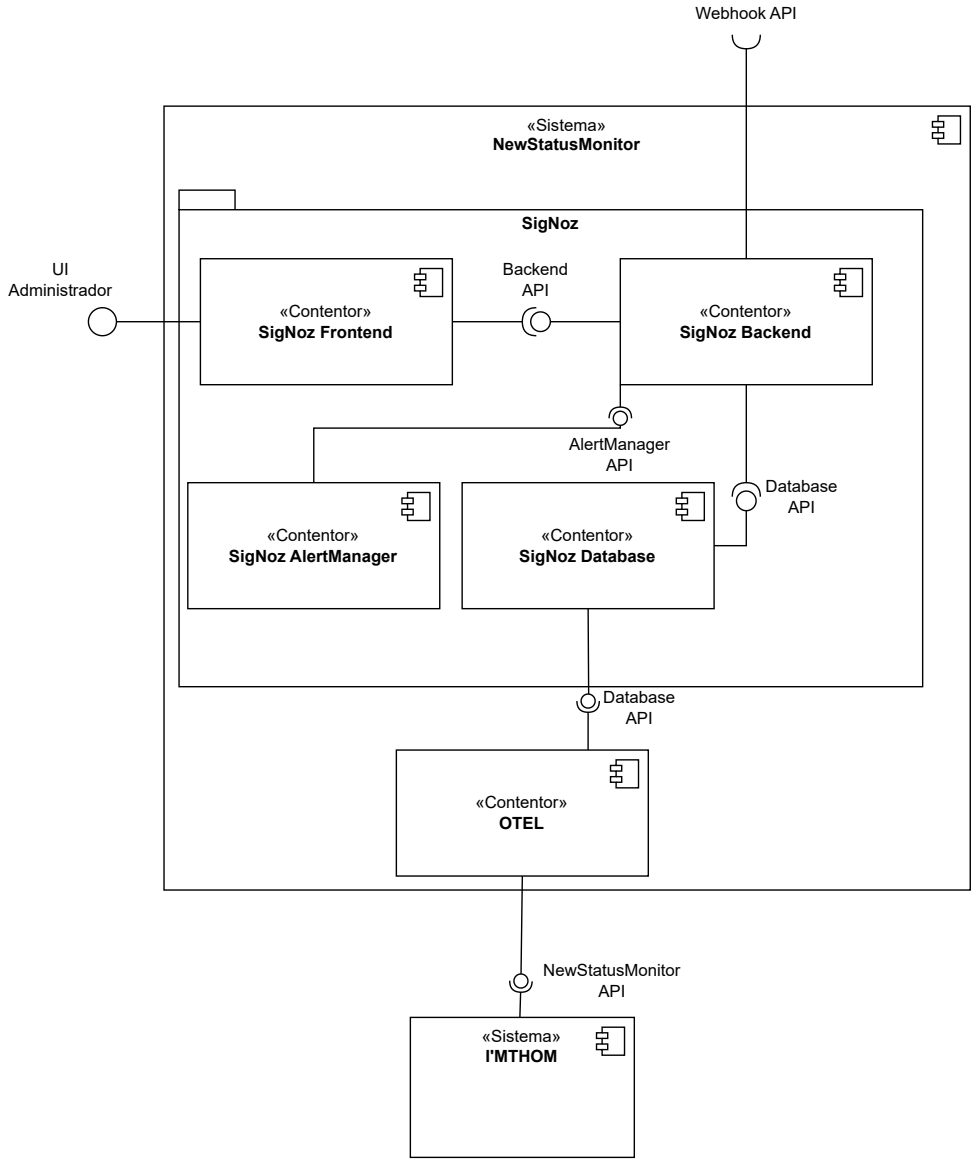

Figura 4.2: Vista Lógica de Nível 2.

Como se pode observar, o sistema SigNoz - contentor principal do sistema - é composto por 3 (três) camadas, sendo elas [46]:

- **Camada de Apresentação**: Diz respeito à interface, seja ela gráfica ao não, disponibilizada ao utilizador final para que este consiga comunicar com o sistema;
- **Camada de Aplicação**: Diz respeito à camada de negócio da aplicação. É responsável pelo processamento dos dados provenientes da camada mencionada anteriormente, e pelo posterior envio destes para a camada de dados;
- **Camada de Dados**: Diz respeito à camada de infraestrutura da aplicação. É responsável pelo armazenamento e fornecimento de dados sempre que estes forem requisitados.

Tendo tais definições como base, é possível fazer o paralelismo para os contentores do sistema SigNoz:

- **SigNoz Frontend**: Enquadra-se na camada de apresentação. É responsável por disponibilizar uma interface gráfica *web* aos utilizadores;
- **SigNoz Backend**: Enquadra-se na camada de aplicação. É responsável por receber os dados telemétrico provenientes do contentor OTEL, processá-los e posteriormente enviá-los para o contentor SigNoz Database. É também responsável por disponibilizar uma API que é utilizada pelo contentor SigNoz Frontend;
- **SigNoz Database**: Enquadra-se na camada de dados. É responsável por armazenar de forma persistente os dados que recebe do contentor SigNoz Backend e posteriormente disponibilizá-los ao mesmo quando este os solicita.

4.2.2 Vista Física

A Figura 4.3 apresenta a infraestrutura física que compõe o sistema NewStatusMonitor e como este se relaciona, de uma perspetiva diferente, com o sistema a ser monitorizado:

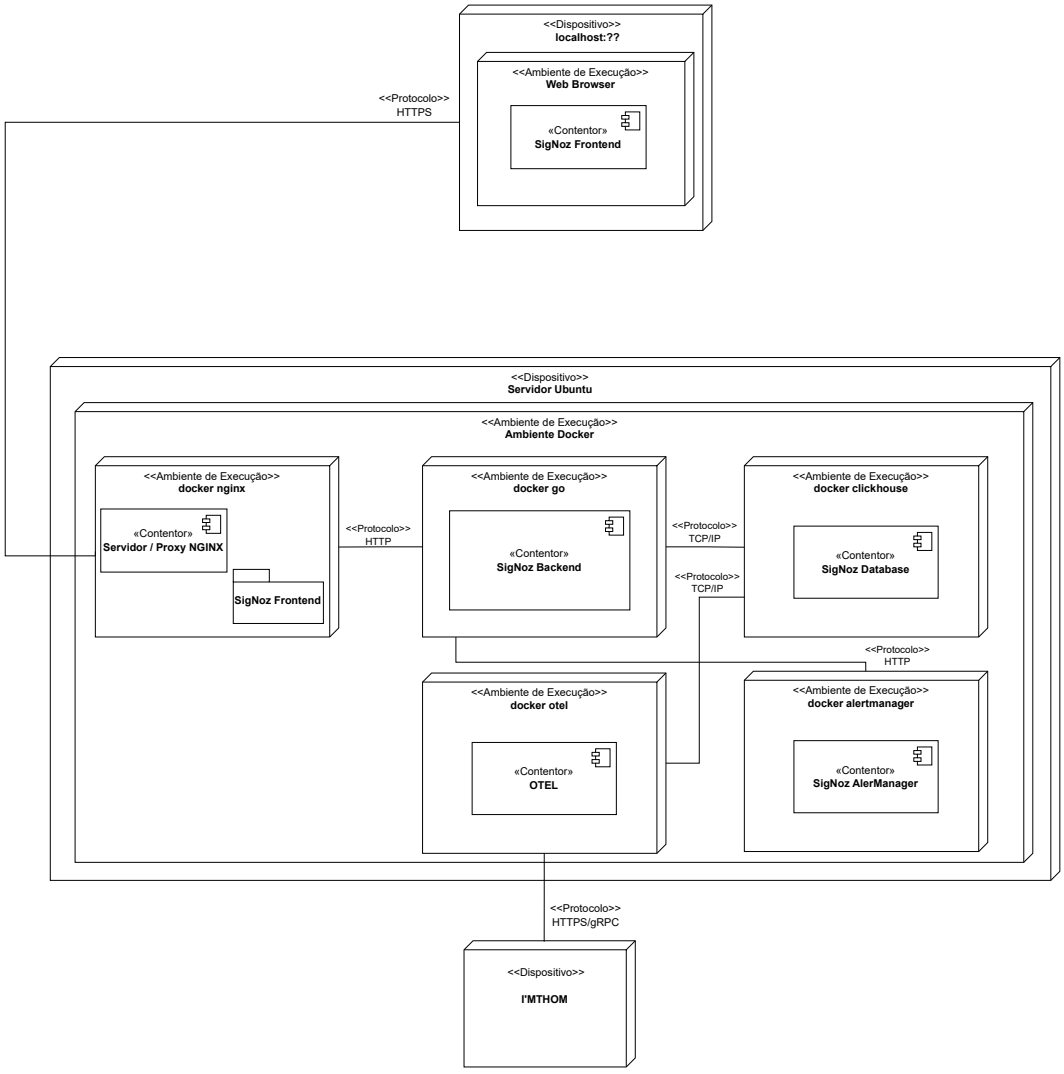

Figura 4.3: Vista Física de Nível 2.

Como é possível observar, a figura apresenta 3 (três) dispositivos físicos. Começando pelo lado esquerdo da mesma, o primeiro dispositivo diz respeito ao próprio navegador *web* do utilizador final, enquanto o segundo diz respeito ao servidor do sistema de monitorização, neste caso um servidor Ubuntu. Neste encontram-se múltiplos contentores Docker que são orquestrados através do uso da ferramenta Docker Compose. Cada contentor presente na subsecção 4.2.1 é mapeado em um contentor nesta vista, à exceção do servidor NGINX, por este não se tratar de um componente lógico do sistema.

É de se notar que todos os contentores foram representados dentro do mesmo dispositivo porque o volume de dados não justifica a separação dos mesmos em ambientes físicos diferentes, contudo tal é possível através de uso de ferramentas como Kubernetes.

Capítulo 5

Implementação

Neste capítulo é abordada a implementação da solução, tendo por base a arquitetura apresentada anteriormente. O mesmo encontra-se dividido nas 3 (três) grandes etapas descritas anteriormente da *pipeline* de observabilidade pretendida. Para cada secção, são expostas as decisões técnicas tomadas, mencionando as principais tecnologias utilizadas no desenvolvimento do projeto.

5.1 Instrumentação

Relativamente ao processo de instrumentação do I'MTHOM, como já referido anteriormente na secção 1.2, esta aplicação já possuía previamente instrumentação própria desenvolvida e mantida pela própria empresa proponente. Contudo, a mesma era incapaz de disponibilizar todas as métricas consideradas relevantes pela empresa proponente. Tendo por base este contexto, foi feita uma pesquisa pelas principais ferramentas FOSS de instrumentação existentes no mercado e como mencionado na secção 2.2, optou-se pela OpenTelemetry.

O I'MTHOM é desenvolvido com recurso à *framework* ASP.NET[1]. Ora, infelizmente, ao momento de escrita deste relatório, o SDK da OpenTelemetry para ASP.NET encontra-se ainda numa fase experimental, o que significa que ainda não possui qualquer lançamento considerado estável pela equipa de desenvolvimento e portanto, não se encontra ainda apta a capturar todas as métricas estipuladas pela especificação da OpenTelemetry, nem é recomendada a sua adoção em ambientes de produção. Isto, porque dado o seu estado atual, a documentação deste SDK é limitada e consequentemente a manutenção de código que faça uso do mesmo torna-se difícil. À data da escrita deste relatório a única documentação existente deste SDK consiste em somente um ficheiro Markdown[2] [47]. Além disso, durante as múltiplas tentativas de integrar este SDK com o projeto I'MTHOM no curto período de tempo que este projeto engloba, houve uma série de mudanças na implementação da API, novamente devido ao facto de se encontrar num estado muito inicial de desenvolvimento, que resultou com que vários métodos fossem descartados e outros criados, por parte dos contribuidores desta *framework* de instrumentação.

É importante, contudo, destacar que mesmo apesar de tais adversidades existirem no presente, a adoção do projeto de OpenTelemetry pode ainda assim ser vista como uma decisão acertada pensando no futuro. Este projeto, como referido anteriormente na subsecção 2.2.1, este projeto é apoiado por uma comunidade ativa com desenvolvedores de múltiplas empresas e organizações líderes no setor de monitorização [47]. Isto permite ter a noção de que existe uma enorme probabilidade de que o mesmo continue o seu desenvolvimento ativa e

[1] https://learn.microsoft.com/en-us/aspnet/overview
[2] https://daringfireball.net/projects/markdown/syntax

que o número de soluções de análise que oferecem suporte ao formato de dados da OpenTelemetry continue a aumentar ao longo do tempo. Pela análise do histórico de versões do projeto de instrumentação da *framework* ASP.NET, o autor deste projeto estima que o mesmo atinja uma versão estável ainda no ano de 2023, o que permitiria que este passasse a ter um nível de maturidade suficiente para ser integrado em produção.

A isto acresce o facto de que, após reuniões com o supervisor da empresa, ficou clara a intenção da Ciberbit migrar o desenvolvimento do I'MTHOM de ASP.NET para ASP.NET Core[3]. Esta *framework*, ao contrário do que acontece com a ASP.NET, tem a sua instrumentação desenvolvida no repositório oficial do projeto OpenTelemetry [48] para .NET, que por sua vez, à data de escrita deste relatório, já se encontra estável com a versão 1.4.0 a permitir o uso da mesma em produção.

Em suma, o desenvolvimento de instrumentação com recurso ao projeto OpenTelemetry, seja através do SDK de ASP.NET quando este chegar à maturidade, seja através do SDK oficial da OpenTelemetry para ASP.NET Core é uma decisão, na opinião do autor deste relatório, positiva, uma vez que garante a robustez e adaptabilidade necessárias para instrumentar corretamente o I'MTHOM como também a flexibilidade na hora de selecionar uma ferramenta de análise, caso necessário, no futuro. Além de que a longo prazo, o processo de manter um sistema de telemetria por conta própria, que é que atualmente existe na Ciberbit, tende a ser muito mais demoroso e a necessitar de mais recursos humanos [49].

Ora, de modo a se mostrar o resto da *pipeline* de observabilidade em funcionamento, doravante é feito uso de uma aplicação de simulação simplista cujo propósito, como o próprio nome já indica, passa por simular o funcionamento de uma aplicação distribuída. Esta aplicação faz uso entretanto de tecnologias passíveis de serem instrumentadas com recurso aos SDKs do projeto OpenTelemetry. No caso, a mesma foi desenvolvida recorrendo a NodeJS[4], devido à familiaridade do autor com a mesma, bem como à vasta gama de exemplos e completa documentação para a integração da mesma com o ecossistema do projeto OpenTelemetry disponível na internet. Não foi feito uso da atual instrumentação do I'MTHOM por esta possuir uma estruturação própria não padronizada e portanto, não compatível com as especificações impostas pela OpenTelemetry.

5.2 Processamento

Relativamente à etapa de processamento dos dados telemétricos, esta foi divida em duas partes. Em termos de tecnologias escolhidas, como já ilustrado na subsecção 4.2.2, a primeira parte diz respeito ao uso do OTEL e a segunda ao uso da ClickHouse.

5.2.1 OTEL

Como já referido na subsecção 2.2.1, a adoção do OTEL permite criar uma camada de abstração entre a aplicação monitorizada e a ferramenta de análise. Este é capaz de receber dados telemétricos numa variedade enorme de formatos, aplicar-lhes algum tipo de processamento lógico e posteriormente exportar o resultado também numa variedade de formatos. Desta forma, basta-nos instrumentar a nossa aplicação apenas uma vez mesmo que no futuro se considere a migração para uma outra ferramenta de análise, ou seja, em

[3] https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core
[4] https://nodejs.org/

suma este componente serve o único propósito de desacoplar a nossa instrumentação da nossa ferramenta de análise.

Dito isto, seguem-se no Código 5.1 os detalhes deste serviço, presentes no ficheiro *docker-compose.yaml* produzido:

```
otel-collector:
  image: signoz/signoz-otel-collector:0.76.1
  volumes:
    - ./config/opentelemetry_collector/opentelemetry-collector-config.yaml
    :/etc/otel-collector-config.yaml
  command: ["--config=/etc/otel-collector-config.yaml", "--feature-gates=-
    pkg.translator.prometheus.NormalizeName"]
  environment:
    - OTEL_RESOURCE_ATTRIBUTES=host.name=ciberbit-host,os.type=linux
    - DOCKER_MULTI_NODE_CLUSTER=false
    - LOW_CARDINAL_EXCEPTION_GROUPING=false
  ports:
    - "4317:4317"
    - "4318:4318"
  restart: on-failure
  depends_on:
    clickhouse:
      condition: service_healthy
```

Código 5.1: Configuração do OTEL.

A partir da observação do mesmo, é possível tirar alguns apontamentos:

- Este serviço faz uso da versão 0.76.1 do OTEL produzido pela própria equipa do SigNoz. Optou-se por selecionar esta implementação do OTEL uma vez que o mesmo possui 2 (dois) *exporters* que facilitam a integração com a ClickHouse. Contudo a implementação presente em [50] poderia também ser utilizada;
- É criado um mapeamento entre o ficheiro *opentelemetry-collector-config.yaml* na máquina *host* e o contentor, de modo a que as alterações em um se reflitam no outro;
- São expostas duas portas do contentor para o exterior, por forma a ser capaz de receber dados telemétricos;
- São criadas 3 (três) variáveis de ambiente que são passadas para o contentor por forma a serem utilizadas aquando da configuração inicial do serviço. Estas são discutidas mais à frente;
- Este serviço deve aguardar que a ClickHouse se encontre ativa e em estado saudável antes de iniciar os trabalhos, uma vez que a ClickHouse é o seu único *exporter* e portanto, o OTEL depende inerentemente da mesma.

Apresentado o contentor que hospeda o OTEL, é feita agora uma análise do ficheiro *opentelemetry-collector-config.yaml* que configura e suporta o comportamento do mesmo. Como referido anteriormente na subsecção 2.2.1, este serviço encontra-se dividido em 3 (três) grandes partes: *receivers*, *processors* e *exporters*. Segue-se no Código 5.2, o excerto YAML[5] relativo aos *receivers*:

[5] https://yaml.org/spec/

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318
        cors:
          allowed_origins:
            - "http://*"
            - "https://*"
  otlp/spanmetrics:
    protocols:
      grpc:
        endpoint: localhost:12345
  hostmetrics:
    collection_interval: 30s
    scrapers:
      cpu: {}
      load: {}
      memory: {}
      disk: {}
      filesystem: {}
      network: {}
```

Código 5.2: *Receivers* do OTEL

A partir da observação do mesmo, é possível tirar algumas conclusões sobre estes *receivers*:

- Este OTEL encontra-se configurado para disponibilizar 2 (dois) *endpoints* em 2 (duas) portas distintas. A porta 4317 é responsável por receber dados telemétricos no formato OTLP através do protocolo Google *Remote Procedure Call* (gRPC), enquanto que a porta 4318 é responsável por recebê-los no mesmo formato, mas através do protocolo *HTTP* provenientes de qualquer localização;
- O *otlp/spanmetrics*, apesar de também disponibilizar uma porta, não recebe qualquer tipo de dados telemétricos. Apenas é colocado aqui, devido ao facto da especificação do OTEL obrigar a que todas as *pipelines* internas façam referência, pelo menos, um *receiver*. Pode ser visto como um *dummy receiver*;
- À semelhança do anterior, o *hostmetrics receiver* não tem como função receber dados telemétricos das aplicações monitorizadas, mas sim capturar métricas relativas ao ambiente em que o OTEL se encontra implantado, ou seja, a máquina física onde o contentor está a correr. Estas métricas são obtidas através da análise do sistema de ficheiros e atualmente apenas se encontra disponível em sistemas Linux, neste projeto através do uso de *Windows Subsystem for Linux* (WSL).

É de realce mencionar que para se fazer uso do *receiver hostmetrics* corretamente, o OTEL deve ser executado como um agente, ou seja, deve ser implantado na mesma infraestrutura física da aplicação que se pretende monitorizar. Quando necessário monitorizar múltiplas máquinas, deve ser implantado um OTEL em cada uma das mesmas e posteriormente os dados telemétricos capturados pelos agentes são enviados para um OTEL central com uma infraestrutura, se possível, própria.

No que diz respeito aos *processors*, o Código 5.3 apresenta os detalhes dos mesmos:

```
processors:
  batch:
    send_batch_size: 5000
    send_batch_max_size: 10000
    timeout: 15s
  signozspanmetrics/prometheus:
    metrics_exporter: otlp/spanmetrics
    latency_histogram_buckets: [1ms, 10ms, 100ms, 250ms, 500ms, 1000ms,
    2000ms, 5s, 10s, 20s, 40s, 60s]
    dimensions_cache_size: 3000
    aggregation_temporality: "AGGREGATION_TEMPORALITY_CUMULATIVE"
    metrics_flush_interval: 30s
    dimensions:
      - name: http.method
        default: GET
      - name: http.status_code
      - name: "ciberbit.collector.one"
  resourcedetection:
    detectors: [system, env]
    timeout: 5s
```

Código 5.3: *Processors* do OTEL

Através deste é importante destacar alguns pontos sobre os mesmos, tais como:

- O *batch processor* é utilizado para agrupar métricas e *traces* em blocos de 5000 sinais (no máximo, 10000) para uma eficiência de transmissão. No caso de não conseguir capturar 10000 sinais no intervalo de tempo de 15 segundos, é feito o envio do bloco existente naquele momento em específico;
- O *signozspanmetrics/prometheus* tem o propósito de criar métricas a partir dos *traces* capturados. Algumas destas métricas criadas automaticamente incluem a duração de um pedido e o número de pedidos para um determinado microsserviço da aplicação. As dimensões apresentadas são adicionadas a algumas dimensões já criadas por padrão e consistem em metadados que ajudam a produzir as métricas, por exemplo, por forma a ajudar o OTEL a ser capaz de caracterizar o número de pedidos recebidos pelos diferentes componentes da aplicação;
- O *resourcedetection processor* é responsável por adicionar metadados relevantes às métricas recolhidas por norma a caracterizar a sua origem, ou seja, o recurso (do inglês *resource*) que os produziu. No contexto deste projeto são configurados 2 (dois), o primeiro diz respeito ao próprio sistema onde o OTEL está implantado, enquanto o segundo refere-se à variável de ambiente *OTEL_RESOURCE_ATTRIBUTES* anteriormente definida no Código 5.1. Na prática, o OTEL tenta analisar o sistema de forma a obter metadados relevantes do recurso, por exemplo, o *hostname* e o sistema operativo do mesmo por si mesmo durante o *timeout* configurado, neste caso 5 (cinco) segundos. Caso tal não seja possível, os dados da variável de ambiente são utilizados.

Feita a recolha e aplicado o processamento dos dados telemétricos capturados, resta então exportar o resultado para a ferramenta de análise selecionada. Segue-se no Código 5.4, os detalhes técnicos produzidos para representar os *exporters*:

```
exporters:
  clickhousetraces:
    datasource: tcp://clickhouse:9000/?database=signoz_traces
    docker_multi_node_cluster: ${DOCKER_MULTI_NODE_CLUSTER}
    low_cardinal_exception_grouping: ${LOW_CARDINAL_EXCEPTION_GROUPING}
  clickhousemetricswrite:
    endpoint: tcp://clickhouse:9000/?database=signoz_metrics
    resource_to_telemetry_conversion:
      enabled: true
  otlp/spanmetrics:
    endpoint: "localhost:4317"
    tls:
      insecure: true
```

Código 5.4: *Exporters* do OTEL

Através da leitura do mesmo, é possível perceber que são utilizados 3 (três) *exporters*, 2 (dois) deles fazendo uso do mesmo destino, com configurações diferentes. Como já mostrado na subsecção 4.2, pela forma como o fluxo que sustenta as ações do SigNoz se encontra desenvolvido, os dados telemétricos são diretamente enviados para a camada de persistência e posteriormente consultados pela Backend. O terceiro *exporter* não exporta nada, na verdade. O que acontece prática é que após a criação das métricas anteriormente descritas, as mesmas são enviadas sob o formato OTLP e através do protocolo gRPC novamente para o OTEL. Dito isto, vamos então, detalhar um pouco a configuração dos 2 (dois) primeiros:

- O primeiro *exporter* é responsável por enviar os *traces* processados para uma base de dados criada especificamente para persistir este tipo de dados. É feito uso de 2 (duas) variáveis de ambiente definidas anteriormente no Código 5.1. A primeira informa se estamos a usar múltiplos nós da nossa camada persistência, enquanto que a segunda diz respeito ao agrupamento de exceções capturadas pelo nome do serviço e o tipo de exceção e é realizado por forma a reduzir a cardinalidade;
- À semelhança do anterior, este é da mesma forma responsável por enviar as métricas processadas para uma base de dados criada especificamente para persistir este tipo de dados. A opção *resource_to_telemetry_conversion* faz com que os atributos dos recursos sejam transformados em *labels* associadas às métricas capturadas.

Ora, explicadas as 3 (três) camadas de funcionamento do OTEL produzido individualmente, segue-se no Código 5.5 a especificação das *pipelines* internas do OTEL que as ligam e permitem ter um melhor entendimento do que realmente acontece dentro deste serviço:

```
service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [clickhousetraces]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [clickhousemetricswrite]
    metrics/generic:
      receivers: [hostmetrics]
```

```
      processors: [resourcedetection, batch]
      exporters: [clickhousemetricswrite]
    metrics/spanmetrics:
      receivers: [otlp/spanmetrics]
      exporters: [otlp/spanmetrics]
```

Código 5.5: Fluxo do OTEL

Como é possível observar, foram criadas 4 (quatro) *pipelines* internas de forma a conectar os componentes anteriormente descritos. De forma bastante sucinta, este excerto de código explicita o fluxo que os dados telemétricos percorrem dentro do OTEL, isto é, através de que mecanismos são capturados, que tipo de lógica lhes é aplicado e de que forma são posteriormente distribuídos. Os 3 (três) primeiros fluxos fazem todos uso de *processors* de forma explicita, enquanto que o último faz uso implícito do *processor* cujo nome é igual ao *receiver* e *exporter*.

Por fim, durante o desenvolvimento desta *pipeline*, apenas foi feito uso de uma instância *standalone* do OTEL. Contudo, dado o volume de dados da Ciberbit, em ambiente de produção é recomendado usar-se, pelo menos, 2 (duas) instâncias seriam recomendadas. Uma parte lidar com *traces* e métricas e outras somente com métricas [35]. Os ficheiros produzidos, tanto o próprio *docker-compose.yaml* como o *clickhouse-cluster.xml* possuem as alterações necessárias, mas comentadas.

5.2.2 ClickHouse

Ora, após o envio dos dados por parte do OTEL, resta-nos então persisti-los. Tendo esta premissa por base, optou-se pela ClickHouse[6] para ser a base de dados responsável por armazenar os dados telemétricos. Uma outra base de dados considerada foi a Apache Druid[7]. Estas bases de dados são ambas estruturadas por colunas, o que permite com que os processos de agregação realizados sob os dados telemétricos sejam muito mais eficientes [51]. Desenvolvida inicialmente pela Yandex[8], a ClickHouse foi a escolhida por ser capaz de iniciar as suas operações num tempo mais curto e também por consumir menos memória quando em produção, uma vez que é desenvolvida em C++ ao contrário da Apache Druid que é desenvolvida em Java e faz uso do *Garbage Collector* (GC) [52]. De forma sucinta, ambas as bases de dados possuem os seus pontos fortes e fracos e qualquer uma das duas seria uma boa escolha para o contexto deste projeto, contudo dado o excelente desempenho apresentado pela ClickHouse e os possíveis problemas que podem surgir quando o volume de dados aumenta em bases de dados baseadas em linguagens de programação com GC embutido, a mesma foi a selecionada [53] [54].

Dito isto, segue-se no Código 5.6, o excerto do *docker-compose.yaml* relativo à ClickHouse:

```
clickhouse:
  image: clickhouse/clickhouse-server:22.8.8-alpine
  tty: true
  depends_on:
    - zookeeper-1
  logging:
    options:
```

[6] https://clickhouse.com/
[7] https://druid.apache.org/
[8] https://yandex.com/dev/clickhouse/

```
      max-size: 150m
      max-file: "3"
  healthcheck:
    test: ["CMD", "curl", "-sfSI", "localhost:8123/ping"]
    interval: 60s
    timeout: 15s
    retries: 5
  ulimits:
    nproc: 65535
    nofile:
      soft: 262144
      hard: 262144
  container_name: clickhouse
  hostname: clickhouse
  ports:
    - "8123:8123"
    - "9000:9000"
  volumes:
    - ./config/clickhouse/clickhouse-config.xml:/etc/clickhouse-server/
    config.xml
    - ./config/clickhouse/clickhouse-users.xml:/etc/clickhouse-server/
    users.xml
    - ./config/clickhouse/custom-function.xml:/etc/clickhouse-server/
    custom-function.xml
    - ./config/clickhouse/clickhouse-cluster.xml:/etc/clickhouse-server/
    config.d/cluster.xml
    - ./config/clickhouse/user_scripts:/var/lib/clickhouse/user_scripts/
    - ./data/clickhouse/:/var/lib/clickhouse/
  restart: on-failure
```

Código 5.6: Configuração da ClickHouse.

A partir deste, é possível destacar alguns pontos mais relevantes, tais como:

- Este serviço depende de um outro designado por *zookeeper-1* que é descrito mais à frente no relatório.
- É feito o mapeamento de 2 (duas) portas para o exterior. A porta 9000 como já mencionado na subsecção anterior é responsável por receber as comunicações por parte do OTEL, neste contexto. A porta 8123 é exposta por forma a disponibilizar a API HTTP desta base de dados;
- É o feito o mapeamento entre vários ficheiros da máquina *host* e o contentor, através da criação de volumes. Estes ficheiros são responsáveis por configurar a instância da ClickHouse;
- Através do uso de *ulimits*, é possível controlar o uso de recursos por parte do contentor em questão. A opção *nproc* diz respeito ao número de processos permitidos, enquanto que a opção *nofile* diz respeito ao número de ficheiros. Após análise de vários ficheiros *docker-compose.yaml* disponibilizados no GitHub[9], o autor considerou estes valores como aceitáveis para este projeto.

Posteriormente, é feito uso da ferramenta Apache Zookeeper[10] como sistema de coordenação da ClickHouse, isto é, serve o propósito de gerir as instâncias de base de dados tendo como base a replicação dos dados por forma a disponibilizar garantias de *failover*. Esta

[9] https://github.com/
[10] https://zookeeper.apache.org/

ferramenta é um dos sistemas FOSS de coordenação mais utilizados no momento e era até há bem pouco tempo o sistema recomendado pela própria equipa de desenvolvimento da ClickHouse [55] [56]. Tendo por base o Zookeeper, a Yandex começou o desenvolvimento de uma solução própria designada por ClickHouse Keeper e no momento de escrita deste relatório, é o sistema de coordenação recomendado pela mesma [57]. Contudo, por facilidade de uso, maturidade do projeto e suporte pela equipa *core* do SigNoz, o autor optou pelo Apache Zookeeper para desenvolver a *pipeline* de observabilidade [58].

Assim sendo, em seguida é apresentado no Código 5.7 o excerto do *docker-compose.yaml* deste serviço:

```
zookeeper:
  image: bitnami/zookeeper:3.7.1
  container_name: zookeeper
  hostname: zookeeper
  ports:
    - "2181:2181"
    - "2888:2888"
    - "3888:3888"
  volumes:
    - ./data/zookeeper-1:/bitnami/zookeeper
  environment:
    - ZOO_SERVER_ID=1
    - ALLOW_ANONYMOUS_LOGIN=yes
    - ZOO_AUTOPURGE_INTERVAL=1
```

Código 5.7: Configuração do Apache Zookeeper.

A partir da leitura do mesmo, podemos observar que este serviço faz expõe 3 (três) portas com a máquina *host*. A porta 2181 é a porta pré-definido do Apache Zookeeper para conexões por parte de clientes, neste caso da ClickHouse. De forma geral, a porta 2888 é utilizada para os múltiplas instâncias do Zookeeper comunicarem entre si, enquanto que a porta 3888 é utilizada para eleger um líder que irá coordenar o processo de replicação de dados. A distinção entre este nó e os restantes é dada pela variável de ambiente *ZOO_SERVER_ID*.

Para finalizar, como foi possível observar, à semelhança do que aconteceu com o OTEL, durante o desenvolvimento desta *pipeline* apenas foi utilizada uma instância da ClickHouse e do Apache Zookeeper. Contudo é importante realçar que em um ambiente de produção é recomendado utilizar 3 instâncias para ambos os componentes [35] [59].

5.3 Visualização

Após o processamento dos dados telemétricos, resta a parte de visualização. Para isso, como já referido anteriormente na subsecção 2.2.5, foi feito uso do SigNoz. Apesar de se tratar de uma solução de observabilidade completa, a parte de visualização do mesmo pode ser dividida em dois serviços, arquiteturalmente ilustrados na subsecção 4.2.2, a Backend e a Frontend.

Ora, à semelhança do que acontecue com os restantes serviços, estes 2 (dois) também foram enquadrados em um ficheiro *docker-compose.yaml*. Segue-se no Código 5.8, os detalhes da Backend:

```
query-service:
  image: signoz/query-service:0.19.0
  container_name: signoz-backend
  command: ["-config=/root/config/prometheus.yml"]
  ports:
    # - "8089:8080" # Public Port
  volumes:
    - ./config/query_service/prometheus.yaml:/root/config/prometheus.yml
    - ./config/alertmanager/alertmanager.yaml:/root/config/alertmanager.
    yml
    - ./data/dashboards:/root/config/dashboards
    - ./data/signoz/:/var/lib/signoz/
  environment:
    - ClickHouseUrl=tcp://clickhouse:9000/?database=signoz_traces
    - ALERTMANAGER_API_PREFIX=http://alertmanager:9093/api/
    - SIGNOZ_LOCAL_DB_PATH=/var/lib/signoz/signoz.db
    - DASHBOARDS_PATH=/root/config/dashboards
    - STORAGE=clickhouse
    - GODEBUG=netdns=go
    - TELEMETRY_ENABLED=true
    - DEPLOYMENT_TYPE=docker-standalone-amd
  restart: on-failure
  healthcheck:
    test: ["CMD", "wget", "--spider", "-q", "localhost:8080/api/v1/health"
    ]
    interval: 30s
    timeout: 5s
    retries: 3
  depends_on:
    clickhouse:
      condition: service_healthy
```

Código 5.8: Configuração da Backend.

A partir deste, é importante destacar alguns pontos ao leitor:

- É feito o mapeamento entre múltiplos ficheiros da máquina *host* e o contentor. O ficheiro *prometheus.yaml* que é passado como argumento no comando que levanta Backend, é responsável por conter as configurações necessárias à comunicação deste serviço com o AlertManager. É também neste que é explicitado a localização das métricas a serem utilizadas, no caso, é feita uma referência à base de dados responsável por persistir as métricas, anteriormente descrita. Por sua vez, o ficheiro *alertmanager.yaml* é discutido mais à frente na secção 5.4. Além destes ficheiros, é importante realçar a pasta *dashboards* que é responsável por persistir tanto os *dashboards* criados através da interface gráfica do SigNoz como outros criados em outras plataformas - por exemplo, Grafana - e posteriormente importados através do uso de ficheiros JSON;
- São passadas múltiplas variáveis de ambiente ao contentor, por forma a ajustar o comportamento da Backend. As mais relevantes acabam por ser o mecanismo de persistência que estamos a adotar e devido a estarmos a usar a ClickHouse, a variável *ClickHouseUrl* é criada por forma a informar a Backend da existência da mesma, bem como o *endpoint* do AlertManager;
- É também criado um *healthcheck*. Neste caso, a ferramenta wget[11] para testar se o

[11] https://www.gnu.org/software/wget/

contetor responde a pedidos na porta 8080 de 30 em 30 segundos, com um *timeout* aceitável de 5 (cinco) segundos, onde são efetuadas 3 (três) tentativas. A opção *–spider* por forma a não ser necessário fazer *download* da página toda e desta forma reduzir o uso de *bandwidth*.

Feito uma breve descrição da implantação realizada para a Backend, vamos então proceder à descrição da Frontend que, dito de forma grosseira, é o componente que realmente permite a visualização dos dados telemétricos desta *pipeline*. O Código 5.9 apresenta a descrição técnica do contentor que suporta este serviço:

```
frontend:
  image: signoz/frontend:0.19.0
  container_name: signoz-frontend
  restart: on-failure
  depends_on:
    - alertmanager
    - query-service
  ports:
    - "3301:3301"
  volumes:
    - ./config/nginx/nginx-config.conf:/etc/nginx/conf.d/default.conf
  - ./config/nginx/letsencrypt/ssl/:/etc/nginx/ssl/
```

Código 5.9: Configuração da Frontend.

Este serviço não possui muitos detalhes relevantes que já não tenham sido abordados anteriormente para os outros contentores. Apesar de não ser de obrigatória utilização, o único ponto relevante que o autor considera é o mapeamento realizado para as configurações do NGINX, incluindo os certificados *Secure Sockets Laye* (SSL). Este ficheiro além de servir o conteúdo disponibilizado pela própria Frontend, também faz o roteamento dos pedidos da Backend e do AlertManager. Segue-se no Código 5.10, um excerto que mostra a configuração presente nesse ficheiro responsável pelo processo:

```
server {

  ...

  location /api/alertmanager {
    proxy_pass http://alertmanager:9093/api/v2;
    proxy_redirect     off;
    proxy_intercept_errors on;
    proxy_set_header   Host $host;
    proxy_set_header   X-Real-IP $remote_addr;
    proxy_set_header   X-Forwarded-For $remote_addr;
    proxy_set_header   X-Forwarded-Host $server_name;
    proxy_set_header   X-Forwarded_Proto https;
  }

  location /api {
    proxy_pass http://query-service:8080/api;
    proxy_redirect     off;
    proxy_intercept_errors on;
    proxy_set_header   Host $host;
    proxy_set_header   X-Real-IP $remote_addr;
    proxy_set_header   X-Forwarded-For $remote_addr;
```

```
    proxy_set_header   X-Forwarded-Host $server_name;
    proxy_set_header   X-Forwarded_Proto https;
    proxy_read_timeout 60s;
  }

  location / {
    if ( $uri = '/index.html' ) {
      add_header Cache-Control no-store always;
    }
    root   /usr/share/nginx/html;
    index  index.html index.htm;
    try_files $uri $uri/ /index.html;
  }
}
```

Código 5.10: Configuração do NGINX.

A partir deste, podemos destacar cada um dos 3 (três) blocos *location* presentes:

- O primeiro bloco define uma *proxy* versa. Na prática faz o roteamento do tráfego do *endpoint* público para o endereço privado A este roteamento, acresce uma série de *headers* HTTP [35];
- O segundo bloco adota o mesmo comportamento que o primeiro, mas para o serviço AlertManager. Além das opções utilizadas no bloco anterior, acrescentou-se a opção *proxy_read_timeout* que informa o NGINX que este deve fechar a conexão quando não existe qualquer transação de dados durante 60 segundos em opções de leitura consecutivas.
- O último bloco é responsável por servir o *website*, ou seja, a Frontend propriamente dita para todos os outros pedidos. Não é feito uso de *cache*, por forma a mostrar sempre a versão mais recente do serviço.

A título de exemplo, segue-se agora na Figura 5.1, a apresentação de um *trace* gerado pela aplicação de simulação desenvolvida, processado e agora apresentado de forma visual na interface gráfica do SigNoz:

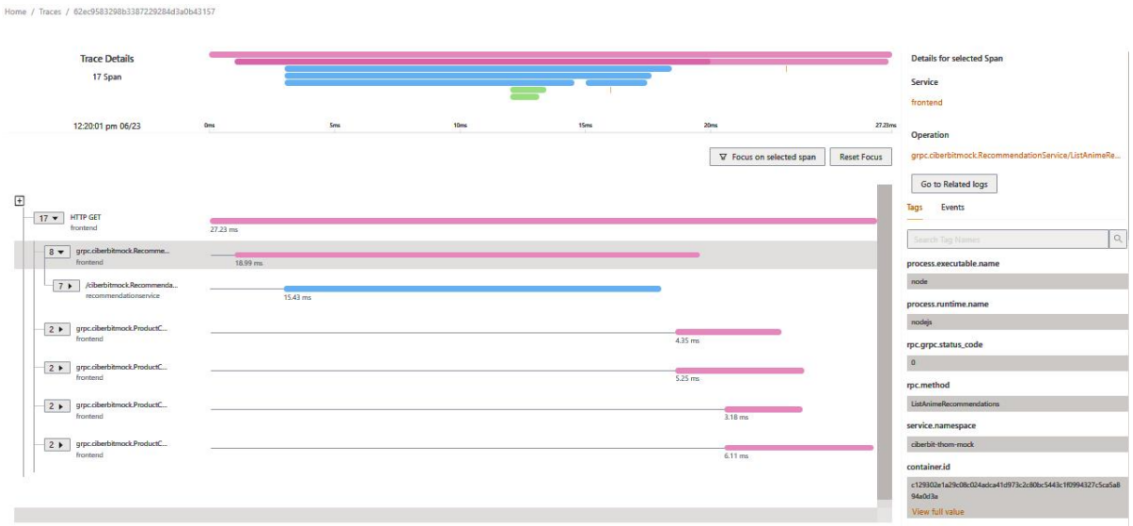

Figura 5.1: Visualização de um *trace* no SigNoz.

Da mesma forma, segue-se agora a título de exemplo a apresentação da métrica *system_cpu_load_average_5m* no SigNoz na Figura 5.2:

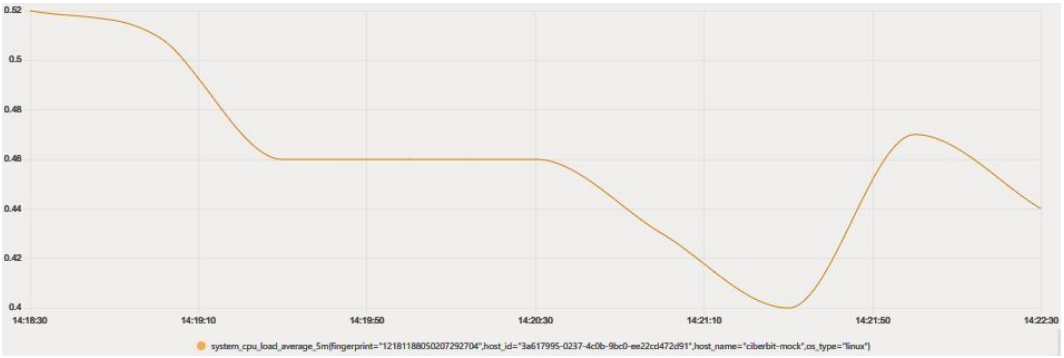

Figura 5.2: Visualização de uma métrica no SigNoz.

5.4 Validação

Produzida a *pipeline* de observabilidade, resta validar o comportamento da mesma. Para tal, foi utilizado um sistema de alarmística. À semelhança dos serviços anteriores, este é também disponibilizado pela equipa do SigNoz e trata-se de um *patch* aplicado ao sistema de alarmística do Prometheus[12], pelo que o mesmo também poderia ter sido adotado para validar a *pipeline*. Optou-se, no entanto, por adotar o disponibilizado pelo próprio SigNoz, tendo sido o principal critério de escolha o suporte fornecido [35]. Dito isto, segue-se no Código 5.11, os detalhes deste serviço:

```
alertmanager:
  image: signoz/alertmanager:0.23.1
  volumes:
    - ./data/alertmanager:/data
  depends_on:
    query-service:
      condition: service_healthy
  restart: on-failure
  command:
    - --queryService.url=http://query-service:8085
    - --storage.path=/data
```

Código 5.11: Configuração do AlertManager.

Como se pode observar, este serviço faz uso da porta interna disponibilizada pela Backend - 8085. Caso se pretendesse fazer uso do AlertManager do próprio Prometheus, havia a necessidade de se utilizar a porta pública explicita no Código 5.8.

Como mostrado anteriormente no Código 5.8, o ficheiro *alertmanager.yaml* contém os detalhes necessários para enviar notificações através do uso de Webhooks. Tal configuração é agora apresentada no Código 5.12:

[12] https://prometheus.io/docs/alerting/latest/alertmanager/

```
global:
  resolve_timeout: 1m

route:
  receiver: "ciberbit-webhook-0001"

receivers:
  - name: "ciberbit-webhook-0001"
  webhook_configs:
  - send_resolved: true
    url: 'https://redacted.ciberbit.pt/api/redacted'
```

Código 5.12: Canais de comunicação do AlertManager.

Neste caso em específico, todas as notificações são enviadas através de um Webhook, cujo *Uniform Resource Locators* (URL) optou-se por não revelar neste relatório. Este canal de comunicação é facilmente criado através da interface gráfica do SigNoz. A Figura 5.3 mostra o mesmo de forma visual a partir da interface gráfica do SigNoz:

Figura 5.3: Canal de comunicação de alertas.

Um dos pontos desfavoráveis encontrados pelo autor durante a fase de testes com a ferramenta escolhida é que esta não permite a criação de um canal com base no protocolo SMTP de forma visual. É possível, à semelhança do é feito no Código 5.12, configurar manualmente um *receiver* que faça uso deste protocolo, contudo tal canal não é posteriormente visível na interface gráfica, ao contrário do que acontece com o canal Webhook criado.

Criado o canal de envio de notificações, resta criar as regras por forma a espoletar os alertas. A título de exemplo para o leitor, são apresentadas 2 (duas) regras que foram criadas com base em métricas por forma a validar a *pipeline* de observabilidade desenvolvida no Código 5.13:

```
groups:
- name: ciberbit-thom_dev-health
  rules:
    - alert: TooHighCPUUsage
      expr: system_cpu_load_average_5m > 0.6
      labels:
        name: "health_check-cpu_usage"
```

```
      severity: warning
      annotations:
        summary: "High CPU Load."
          description: "Too high CPU usage may be a sign of insufficient
    resources and make process unstable. Consider to either increase
    available CPU resources or decrease the load on the process."

    - alert: TooHighMemoryUsage
      expr: (1 - system_memory_available_bytes / system_memory_total_bytes
    ) > 0.8
      for: 1m
      labels:
        name: "health_check-memory_usage"
      severity: critical
      annotations:
        summary: "High Memory load."
        description: "Too high memory usage may result into multiple
    issues such as degraded performance. Consider to either increase
    available memory or decrease the load on the process."

    ...
```

Código 5.13: Configuração de alertas.

Como pode ser observado, foi criada uma regra focada no uso de CPU e outra na RAM por parte do sistema monitorizado. A primeira regra define o espoletar de um alerta quando o uso do CPU ultrapassa, em média, os 60% nos últimos 5 (cinco) minutos, enquanto a segunda define o espoletar de outro alerta quando o uso de memória primária ultrapassa os 80% no último minuto. Por sua vez, a Figura 5.4 mostra os mesmos alertas presentes na interface gráfica do SigNoz:

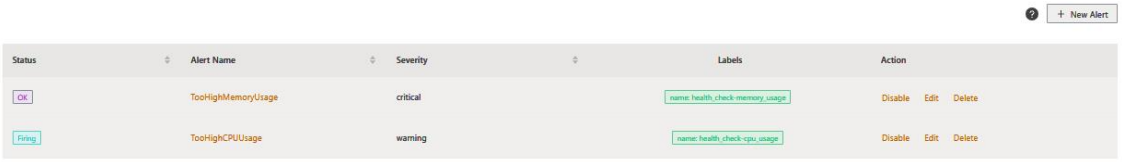

Figura 5.4: Listagem de regras no SigNoz.

Dito isto, todas as regras de alertas produzidas seguem a mesma estrutura base: possuem um nome associado, um grau de severidade, uma condição que deve ser verificada e uma mensagem que é associada aos metadados do alerta.

Ora, por forma a mostrar o espoletar de um alerta, sujeitou-se a máquina do autor a uma situação de estresse de CPU. O Código 5.14 apresenta a mensagem JSON que enviada quando a regra *TooHighCPUUsage* se verificou:

```
{
   "receiver": "ciberbit-webhook-0001",
   "status": "firing",
   "alerts":[
      {
         "status": "firing",
         "labels":{
```

```
            "alertname": "TooHighCPUUsage",
            "fingerprint": "5881192629159634089",
      "fullLabels": "{\"__name__\":\"system_cpu_load_average_1m\",\"
    host_id\":\"ad9d46f8-9ddc-4af5-b9ee-4a05abf90640\",\"host_name\":\"
    ciberbit-mock\",\"os_type\":\"linux\"}",
            "name": "health_check-cpu_usage",
            "ruleId": "2",
            "ruleSource": "Alerting",
            "severity": "warning"
         },
         "annotations":{
            "description": "Too high CPU usage may be a sign of
    insufficient resources and make process unstable. Consider to either
    increase available CPU resources or decrease the load on the process.",
            "summary": "High CPU Load."
         },
         "startsAt": "2023-06-22T15:36:54.925229434Z",
         "endsAt": "0001-01-01T00:00:00Z",
         "generatorURL": "http://alertmanager:9093/graph?g0.expr=
    system_cpu_load_average_5m%20%3E%200.6&g0.tab=1",
         "fingerprint": "b32f2f7bd968f047"
      }
   ],
   "groupLabels": {
      "alertname": "TooHighCPUUsage"
   },
   "externalURL": "http://alertmanager:9093",
   "version": "4",
   "groupKey": "{}/{}:{alertname=\"TooHighCPUUsage\"}",
   "truncatedAlerts": 0
}
```


Código 5.14: Notificação de alerta.

À semelhança do que acontece com as regras anteriormente descritas, esta mensagem JSON é também bastante semelhante para todos os alertas, pelo que autor considerou relevante apenas mostrar 1 (um) exemplo. Ora, em suma, através da criação de regras e posterior verificação da emissão de alertas quando as mesmas acontecem em cenários de testes, foi possível verificar a total integração dos componentes que formam a *pipeline* de observabilidade desenvolvida.

Capítulo 6

Avaliação

Feita a implementação e a implantação da *pipeline* proposta, chega agora o momento de a avaliar. De modo a se concretizar este propósito, vão ser realizados alguns testes de estresse sob a infraestrutura e a aplicação monitorizada de modo a se observar como a *pipeline* de observabilidade se comporta perante estes cenários, nomeadamente, no que diz respeito ao seu desempenho e à sua confiabilidade. Nesta medida, este capítulo descreve inicialmente a abordagem de avaliação adotada, onde é feito um levantamento dos cenários de teste que o autor considerou relevante juntamente com o ambiente onde os testes vão ser realizados. Por fim, o capítulo termina com uma secção onde são apresentados os resultados para cada cenário juntamente com uma breve discussão sobre os mesmos.

6.1 Abordagem

Por forma a avaliar o desempenho da *pipeline* de observabilidade desenvolvida, a abordagem adotada passa por realizar 2 (dois) tipos de teste de desempenho, mais especificamente, testes de estresse e testes de carga, e a partir destes tentar perceber em que medida é que a mesma é capaz de detetar a ocorrência das situações limite impostas e qual o tempo que demora a notificar os utilizadores acerca das mesmas, por forma a satisfazer o atributo de qualidade estipulado na subsecção 3.2.2. Para isso foram estipulados 3 (três) simples cenários que têm por base as métricas inicialmente desejas pela empresa proponente e mencionadas na secção 1.3:

- Uso de CPU acima de 60% durante 1 (um) minuto;
- Uso de RAM acima de 60% durante 1 (um) minuto;
- Número de pedidos recebidos em um *endpoint* específico, por segundo, acima de 35 durante 1 (um) minuto.

Relativamente aos testes de estresse, para induzir o uso de CPU e de RAM foi utilizada a ferramenta HeavyLoad[1]. Esta trata-se de uma ferramenta gratuita que permite realizar facilmente testes de estresse em computadores que utilizem o sistema operativo Windows. Para simular o terceiro cenário, ou seja, realizar o teste de carga, recorreu-se à ferramenta K6[2], ferramenta esta que é FOSS e permite a criação de testes de carga com recurso à linguagem de programação JavaScript de forma simples e compacta.

[1] https://www.jam-software.com/heavyload
[2] https://k6.io/

Dito isto, é importante realçar que este tipo de testes devem ser executados em um ambiente controlado que procure garantir que os resultados obtidos são o mais preciso e fiáveis possível. Contudo, infelizmente, tal não foi passível de acontecer no âmbito deste projeto e portanto, o ambiente de testes resume-se à máquina de trabalho fornecida pela própria empresa proponente cujas especificações são as seguintes:

- Memória: 16GB;
- Número de processadores lógicos: 4;
- Tipo de disco: *Solid-State Drives* (SSD).

Ora, esta máquina foi responsável por executar toda a *pipeline* de observabilidade, bem como a própria aplicação de simulação desenvolvida juntamente com as ferramentas de teste. Excluindo as ferramentas de testes referidas à partida e todos os outros processos nativos do Windows, isto traduz-se, como mostrado no capítulo 5, em 5 (cinco) contentores Docker que suportam a *pipeline* de observabilidade, e outro contentor Docker para suportar a aplicação de simulação. Todos estes contentores são, por sua vez, executados com recurso ao WSL.

Antes de se proceder às experiências e à semelhança do que já foi dito anteriormente, o autor reconhece que a fiabilidade dos resultados pode não ser a ideal e é importante também referir que a abordagem adotada não é de todo uma tentativa de replicar um uso normal do I'MTHOM, mas sim uma forma de avaliar a *pipeline* de observabilidade em condições adversas. Apesar de uma abordagem mais próxima da realidade ser capaz de produzir resultados mais úteis, considerando que se está a utilizar uma aplicação de simulação, tal processo iria necessitar demasiado tempo para desenhar, implementar e só posteriormente então executar testes de estresse e de carga realísticos, pelo que tal abordagem não se verificou.

6.2 Resultados

Cada cenário anteriormente descrito examina o tempo necessário que o sistema de alarmística demora a notificar os utilizadores relativamente ao valor capturado da métrica, quando esta cruza um limiar pré-configurado. A registo temporal começa imediatamente após a condição imposta na regra se verificar na máquina/aplicação. As medidas são feitas de forma manual, pelo que apenas 20 experiências foram realizadas para cada um dos cenários e, desta forma, o autor reconhece que a dimensão da amostra pode ser bastante reduzida.

6.2.1 Cenário de CPU

Por forma, a avaliar o tempo médio que a *pipeline* de observabilidade demora a notificar os utilizadores finais para o uso excessivo de CPU, foi produzida a regra presente no Código 6.1:

```
groups:
- name: chapter8
  rules:
    - alert: TooHighCPUUsage
      expr: system_cpu_load_average_1m > 0.6
      labels:
        name: 'health_check-cpu_usage'
      severity: warning
      annotations:
```

```
        summary: "High CPU Load."
        description: "CH8 - Evaluation Purposes!"
    ...
```

Código 6.1: Regra de avaliação para CPU.

Ora, neste primeiro cenário, Configurou-se a ferramenta HeavyLoad para induzir estresse no CPU em 95%. Entre cada experiência, o alerta foi marcado como resolvido através da interface gráfica do SigNoz e procedeu-se a uma espera de, pelo menos, 3 minutos antes de se realizar a próxima de iteração, de forma a que o uso de CPU diminuísse abaixo do limiar imposto e aí permanecesse durante breves segundos. De modo a respeitar a regra criada, a situação de estresse foi aplicada à máquina durante, pelo menos, 1 minuto. Posteriormente, procedeu-se à contagem dos segundos entre a condição se verificar e a mensagem chegar através do Webhook associado. Dito isto, a Figura 6.1 apresenta os resultados obtidos:

Figura 6.1: Tempo médio de recebimento de alerta com base no CPU.

Como é possível observar, os resultados obtidos, à exceção da iteração 13, a *pipeline* foi capaz de notificar o utilizador num tempo considerado aceitável segundo o atributo de qualidade imposto, mesmo tendo em consideração o uso de CPU elevadíssimo que se verificou na infraestrutura onde a *pipeline* executou.

6.2.2 Cenário de RAM

Relativamente ao segundo cenário, de modo a se estimar o tempo que a *pipeline* de observabilidade demora a notificar os utilizadores finais para o uso excesivo de RAM, foi produzida a regra presente no Código 6.2:

```
groups:
- name: chapter8
  rules:
    - alert: TooHighMemoryUsage
```

```
    expr: (1 - system_memory_available_bytes / system_memory_total_bytes
) > 0.8
  for: 1m
    labels:
      name: "health_check-memory_usage"
    severity: critical
    annotations:
      summary: "High Memory load."
      description: "CH8 - Evaluation Purposes!"
  ...
```

Código 6.2: Regra de avaliação para RAM.

Para este cenário, configurou-se a ferramenta HeavyLoad para alocar memória até a memória disponível atingir somente 2GB. À semelhança do que aconteceu no cenário anterior, entre cada experiência, o alerta foi marcado como resolvido através da interface gráfica do SigNoz e procedeu-se a uma espera de, pelo menos, 1 minuto antes de se realizar a próxima de iteração, de forma a que o uso de RAM diminuísse abaixo do limiar imposto e aí permanecesse durante breves segundos. Seguem-se, então, na Figura 6.2 os resultados obtidos:

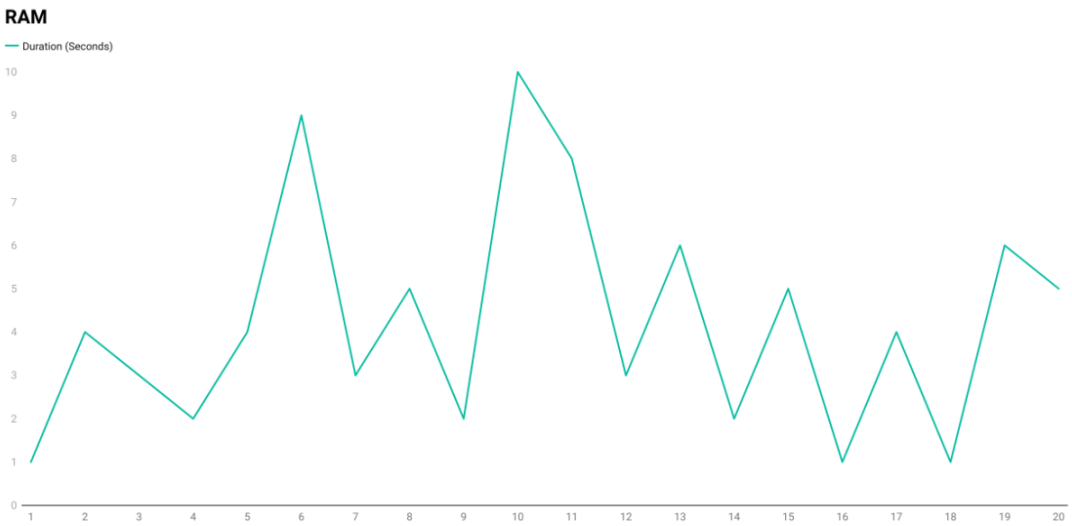

Figura 6.2: Tempo médio de recebimento de alerta em RAM.

Os tempos aqui registados também se encontram abaixo do limite de 15 segundos, à semelhança do primeiro cenário. É possível também observar que os tempos neste cenário acabam por ser inferiores aos apresentados no cenário, o que faz sentido levando em consideração a especificidade da métrica que está a monitorizar neste caso e como os dois componentes de *hardware* influenciam o fluxo de execução de uma aplicação.

6.2.3 Cenário de Requests Per Second (RPS)

Por fim, relativamente ao segundo cenário, de modo a se estimar o tempo que a *pipeline* de observabilidade demora a notificar os utilizadores finais para um número anormal de pedidos HTTP para um *endpoint* específico, foi produzida a regra presente no Código 6.3:

```
groups:
- name: chapter8
  rules:
    - alert: TooHighNetworkTraffic
      expr: rate(http_requests_total{job="animes-catalog", handler="/api/
    cat/anime/product/one_piece"}[1m]) > 35
      labels:
        name: 'health_check-network_usage'
      severity: warning
      annotations:
        summary: "High Network load."
        description: "CH8 - Evaluation Purposes!"
    ...
```

Código 6.3: Regra de avaliação para RPS.

Esta regra avalia se a aplicação monitorizada recebeu no *handler* especificado mais de 35 pedidos por segundo durante o último minuto. Esta métrica é obtida através da análise dos *traces* capturados. Dito isto, para este cenário, configurou-se a ferramenta K6 com 35 utilizadores virtuais, onde cada um destes vai realizar 160 pedidos pausas de um décimo de segundo entre eles, ao longo de 1 (um) minuto. O cenário adotado na ferramenta é apresentado de seguida no Código 6.4:

```
  ...

  scenarios: {
  productOnePiece: {
      executor: "per-vu-iterations",
      exec: "productOnePiece",
      vus: 30,
      iterations: 160,
      maxDuration: '1m',
  }
  }

  ...
```

Código 6.4: Cenário criado no K6.

Através de testes anteriores, o autor observou que a aplicação monitorizada consegue servir, nestas condições, aproximadamente 5100 pedidos por minuto para o *endpoint* indicado, o que resulta em mais de 80 pedidos por segundo e desta forma, suficiente para ativar a regra descrita. Por sua vez, a Figura 6.3 apresenta os tempos registados para o espoletar deste alerta:

Figura 6.3: Tempo médio de recebimento de alerta em RPS.

Como é observável, também neste caso a *pipeline* foi capaz de notificar os utilizadores finais em tempo útil.

6.3 Sumário

Este capítulo determinou que o atributo de desempenho definido anteriormente na subsecção 3.2.2 é atendido pela solução desenvolvida. Estas experiências permitiram também verificar, uma vez mais, a importância da ferramenta de análise adotada se encontrar numa infraestrutura separada da aplicação monitorizada. Dito isto, estes testes de desempenho também permitem à empresa proponente entender os requisitos físicos aceitáveis para suportar a *pipeline* de observabilidade.

Realizando um paralelismo entre esta solução e a ferramenta interna da Ciberbit, o Status Monitor, a primeira foca-se em *traces* e *métricas*, enquanto a segunda apenas lidava com exceções, que se enquadram na catgoria de *logs*. À data de hoje, após a expansão de funcionalidades do mesmo por parte do outro aluno de mestrado, o Status Monitor possui também já as estruturas de dados necessárias para suportar métricas, contudo o I'MTHOM ainda não se encontra devidamente instrumentado para providenciar todas as métricas capazes de atender aos objetivos iniciais. A isto, acresce o facto de a mesma não possuir qualquer referência a *traces*, nem possuir qualquer sistema de alarmística. É de referir ainda que apesar de tal não se verificar no começo do desenvolvimento deste projeto, desde Maio de 2023 que a especificação para transporte de *logs* através de OTLP atingiu o nível estável, contudo devido ao intervalo de tempo reduzido que se verificara, o autor optou por não refletir tal mudança na *pipeline* produzida.

Tendo por base estes resultado, o trabalho descrito anteriormente, bem como o conhecimento adquirido durante este projeto, o capítulo seguinte descreve a opinião do autor em relação à solução final desenvolvida.

Capítulo 7

Conclusão

Este capítulo discute os objetivos concretizados, os não concretizados e o possível trabalho futuro para o projeto em questão. No final é feita uma apreciação final do autor relativamente ao desenvolvimento deste projeto e como este o influenciou.

É importante notar que durante o período de tempo atribuído a este projeto, o objetivo principal do projeto acabou por mudar, o que fez com o que autor tivesse despendido demasiado tempo e recursos de forma desnecessária o que, consequentemente, acabou por influenciar o produtivo final entregue. Inicialmente, o projeto assentava no desenvolvimento de *software*, no caso de uma Backend cujo foco seria tratar dados provenientes de operações monitorização, enquanto mais tarde passou para a implementação e implantação de uma *pipeline* de observabilidade. Apesar disso, o autor focou-se em concretizar este novo objetivo ao máximo, tendo, felizmente, apresentado níveis positivos na avaliação, seja em termos de desempenho como de confiabilidade.

7.1 Objetivos concretizados

Ora, o objetivo principal deste projeto constistiu em aumentar o grau de observabilidade do I'MTHOM, através da captura, processamento e posterior visualiação de *traces* e múltiplas métricas. Por forma a atingir isto, apesar de já existir uma solução de monitorização na empresa proponente, decidiu-se construir uma *pipeline* de observabilidade de raíz.

A solução produzida assenta em soluções FOSS capazes de desacoplar o componente de instrumentação do componente de análise. A camada de processamento construída permite não só a ingestão de elevados volumes de dados, como é também capaz de extrair valor desses mesmos dados, o que não se verificava com a solução interna da Ciberbit. O facto de toda a *pipeline* ter sido construída com recurso a contentores permite uma fácil manutenção ao longo do tempo, bem como uma integração fácil com ferramentas de escalabilidade de renome hoje presentes no mercado, nomeadamente o Kubernetes. Por sua vez, a camada de visualização permite também fornecer uma visualização clara e intuítiva dos dados telemétricos capturados num único lugar através de diversos tipos de visualização nativos, tais como gráficos e listagens. A estes acresce o facto de ser possível criar páneis de visualização costumizados se necessário através do próprio *query builder* gráfico ou através do uso de PromQL. Além disso, o componente de alarmística que foi integrado na *pipeline* provou-se ser uma mais valia dada a sua flexibilidade para atender a diversos casos de negócio. O facto deste também ser passível de integrar na camada de visualização é uma mais valia.

Por estas razões, o autor acredita que as necessidades fundamentais foram atendidas pelo projeto desenvolvido.

7.2 Objetivos não concretizados

O autor considera que os objetivos de maior relevo foram atingidos de forma bem sucedida. Relativamente aos requisitos funcionais, não foi possível produzir corretamente a camada de instrumentação da *pipeline* devido à tecnologia adotada pelo I'MTHOM, bem como devido ao estado de desenvolvimento precoce do SDK da OpenTelemetry para a mesma. De resto, todas as restantes camadas foram corretamente implementadas e implantadas em um ambiente de testes, mesmo que rudimentar. Por sua vez, no que diz respeito aos atributos de qualidade, todos foram levados em consideração pela solução final como é detalhado no Apêndice B.

É importante realçar novamente, que na proposta inicial, criada em finais de outubro de 2022, consistia num projeto diferente. O projeto inicial dizia respeito ao desenvolvimento de uma Backend por parte deste relatório, enquanto que outro aluno de mestrado seria responsável pela criação da Frontend. Contudo, após uma série de reuniões com o orientador, concluiu-se que tal abordagem não era a indicada, pelo que foi decidido por volta de fevereiro de 2023 a abordagem final adotada. Ora, tal alteração levou à reformulação do trabalho produzido, pelo que acabou por prejudicar inerentemente o fluxo de trabalho e sendo o tempo atribuído ao projeto reduzido, isso pode-se ter refletido de certa forma na *pipeline* produzida.

Apesar disso, de uma forma geral, o autor considera que os objetivos atendidos, mesmo que não sejam a totalidade dos propostos, acrescentam valor empresa proponente e permitem criar um caminho que a mesma pode adotar num futuro caso se opte por uma abordagem *Out-Of-The-Box* (OOTB).

7.3 Trabalho futuro

Este projeto, e a solução produzida, ainda possui algumas funcionalidades que podem ser acrescentadas ou melhoradas, tais como:

- Instrumentação do I'MTHOM: por forma a se instrumentar o I'MTHOM com recurso ao projeto OpenTelemetry, existem duas possibilidades. Ou a empresa proponente migra o mesmo de ASP.NET para ASP.NET Core, ou aguarda-se que o SDK da OpenTelemetry para ASP.NET atinja o nível de maturidade necessário para ser utilizado em produção;
- *Logs*: Pegando no primeiro ponto como base, a partir de Maio de 2023, a especificação da OpenTelemetry para *logs* atingiu a estabilidade na sua *Bridge* API, bem como no seu protocolo OTLP, pelo que seria uma mais valia adicionar este pilar à *pipeline* de observabilidade desenvolvida [60];
- Sistema de coordenação: a *pipeline* desenvolvida faz uso do Apache Zookeeper como sistema de coordenação do processo de replicação de dados das instâncias ClickHouse, contudo uma migração do mesmo para o ClickHouse Keeper deve ser considerada num futuro próximo. Uma vez que apesar de o Apache Zookeeper se tratar de uma ferramenta mais madura, o ClickHouse Keeper é o sistema de coordenação recomendável pela própria Yandex e foi desenhado para mitigar alguns dos pontos fracos do Zookeeper;
- Escalabilidade automática: a *pipeline* desenvolvida faz o uso da tecnologia docker-compose que não é capaz de escalar de forma automática mediante as necessidades impostas à mesma, contudo ferramentas como o Kubernetes encontram-se aptas para

realizar este tipo de processos de escalação, pelo que é de recomendável adoção no projeto;

7.4 Apreciação final

Em suma, a solução desenvolvida pode ser vista como um primeiro passo na adoção de uma solução de observabilidade robusta e de fácil manutenção para a Ciberbit, contudo o estado atual não permite que a mesma seja de facto um serviço de monitorização útil à Ciberbit. O autor recomenda que sejam realizadas consultas constantes às ferramentas adotadas na construção da *pipeline*, principalmente, na camada de instrumentação por forma a ter conhecimento quando estes componentes se encontram aptos para produção.

Contudo, fazendo uma comparação com a outra abordagem possível e mencionada na secção 1.3, é de se salientar que a abordagem do autor parece ser a mais promissora a longo prazo, uma vez que, considerando o que já mostrado na secção 6.3, o contexto do problema e a velocidade com que as tecnologias e as suas especificidades mudam, o caminho a seguir para resolver o problema em mãos não parece ser o desenvolvimento de uma plataforma própria e mantê-la ao longo dos anos, mas sim recorrer a soluções robustas, ativamente desenvolvidas e fortemente testadas já existentes no mercado, sejam elas FOSS ou não. A isto acresce o facto de que, a longo prazo, desenvolver e manter uma solução de telemetria por conta própria tende a ser muito mais demoroso e a necessitar de mais recursos humanos [49].

Dito isto, apesar das diversas adversidades encontradas, o autor foi capaz de beneficiar imenso através do desenvolvimento deste projeto. O autor adquiriu bastante experiência e conhecimento sobre a área da observabilidade, bem como as dificuldades que soluções dentro deste contexto enfrentam.

Bibliografia

[1] Ciberbit S.A. *IM'THOM*. Acedido em Jan 2023. url: https://www.imthom.com/.

[2] Emre Ate. *Automating telemetry- and trace-based analytics on large-scale distributed systems*. Ago. de 2020. url: https://open.bu.edu/handle/2144/41472.

[3] Jeffrey Joyce et al. «Monitoring Distributed Systems». Em: *ACM Trans. Comput. Syst.* 5.2 (mar. de 1987), pp. 121–150. issn: 0734-2071. doi: 10.1145/13677.22723. url: https://doi.org/10.1145/13677.22723.

[4] Claire Drumond. *Scrum - what it is, how it works, and why it's awesome*. Jul. de 2022. url: https://www.atlassian.com/agile/scrum.

[5] Devon H. O'Dell. «The Debugging Mindset: Understanding the Psychology of Learning Strategies Leads to Effective Problem-Solving Skills.» Em: *Queue* 15.1 (fev. de 2017), pp. 71–90. issn: 1542-7730. doi: 10.1145/3055301.3068754. url: https://doi.org/10.1145/3055301.3068754.

[6] Sean Peisert, Thomas E. Potok e Todd Jones. «ASCR Cybersecurity for Scientific Computing Integrity - Research Pathways and Ideas Workshop». Em: (jun. de 2015). doi: 10.2172/1236181. url: https://www.osti.gov/biblio/1236181.

[7] Ozan Tuncer et al. «Diagnosing Performance Variations in HPC Applications Using Machine Learning». Em: *Information Security Conference*. 2017.

[8] J. Riedesel. *Software Telemetry: Reliable Logging and Monitoring*. Manning, 2021. isbn: 9781617298141. url: https://amazon.com/Software-Telemetry-Reliable-logging-monitoring/dp/161729814X.

[9] Haoyu Song et al. *Network Telemetry Framework*. RFC 9232. Mai. de 2022. doi: 10.17487/RFC9232. url: https://www.rfc-editor.org/info/rfc9232.

[10] Ata Turk et al. «Seeing into a Public Cloud: Monitoring the Massachusetts Open Cloud». Em: 2016.

[11] Emre Cihan Ates et al. «HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations». Em: *Proceedings of the 48th International Conference on Parallel Processing* (2019).

[12] Xu Zhao et al. «Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold». Em: *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[13] Charity Majors, Liz Fong-Jones e George Miranda. Em: *Observability Engineering: Achieving production excellence*. O'Reilly, 2022.

[14] Raja R. Sambasivan et al. «Principled workflow-centric tracing of distributed systems». Em: *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016).

[15] Jonathan M. Kaldor et al. «Canopy: An End-to-End Performance Tracing And Analysis System». Em: *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[16] Benjamin H. Sigelman et al. «Dapper, a Large-Scale Distributed Systems Tracing Infrastructure». Em: 2010.

[17] Austin Parker et al. *Distributed Tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[18] Yuri Shkuro. *Mastering distributed tracing: Analyzing performance in microservices and Complex Systems*. PACKT Publishing Limited, mar. de 2019.

[19] Alex Boten. *Cloud native observability with OpenTelemetry: Learn to gain visibility into systems by combining tracing, metrics and logging with OpenTelemetry*. PACKT PUBLISHING LIMITED, 2022.

[20] CNCF Contributors. *CNCF - Activity Repository Groups*. 2023. url: `https://all.devstats.cncf.io/d/1/activity-repository-groups?orgId=1`.

[21] B. Brazil. *Prometheus: Up & Running : Infrastructure and Application Performance Monitoring*. O'Reilly Media, 2018. isbn: 9781492034148. url: `https://books.google.pt/books?id=mdv2tQEACAAJ`.

[22] Prometheus Authors. *Exporters and integrations: Prometheus*. Acedido em Mar 2023. url: `https://prometheus.io/docs/instrumenting/exporters/`.

[23] Amazon Authors. *What Is Jaeger?* Acedido em Fev 2023. url: `https://aws.amazon.com/what-is/jaeger/`.

[24] Jaeger Authors. *Features*. 2023. url: `https://www.jaegertracing.io/docs/1.45/features`.

[25] Blueswen. *FASTAPI-Jaeger/readme.md at main ů blueswen/FASTAPI-jaeger*. url: `https://github.com/blueswen/fastapi-jaeger/blob/main/readme.md`.

[26] Hrittik Roy. *Using Jaeger vs. Prometheus*. url: `https://www.airplane.dev/blog/jaeger-vs-prometheus`.

[27] Written byMukhaddin Beshkov. Acedido em Abr 2023. url: `https://www.wallarm.com/cloud-native-products-101/what-is-jaeger`.

[28] Kumar Harsh. *Using SigNoz, an open-source APM*. url: `https://www.airplane.dev/blog/signoz`.

[29] Google. *Go for web development*. Acedido em Mai 2023. url: `https://go.dev/solutions/webdev`.

[30] hanabi1224. *Go vs Python Benchmarks*. Acedido em Mai 2023. url: `https://programming-language-benchmarks.vercel.app/go-vs-python`.

[31] hanabi1224. *Go vs Ruby Benchmarks*. Acedido em Mai 2023. url: `https://programming-language-benchmarks.vercel.app/go-vs-ruby`.

[32] Fev. de 2019. url: `https://news.ycombinator.com/item?id=19131272`.

[33] StackOverflow Contributors. *Most popular technologies*. 2022. url: `https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language`.

[34] Richard Hipp. *Situations Where SQLite Works Well*. url: `https://www.sqlite.org/whentouse.html`.

[35] Estee Wen. *SigNoz - Infrastructure Discussion*. Private Conversation. Abr. de 2023.

[36] Ankit Anand. *Jaeger vs SigNoz*. 2022. url: `https://signoz.io/blog/jaeger-vs-signoz/`.

[37] Mar. de 2023. url: `https://opentelemetry.io/docs/specs/otel/status/#logging`.

[38] SigNoz Authors. *Logs: Signoz*. Mar. de 2023. url: `https://signoz.io/docs/userguide/logs/`.

[39] Dinesh Thakur. *What is software requirement? Types of requirements*. Nov. de 2013. url: `https://ecomputernotes.com/software-engineering/softwarerequirement`.

[40] Lawrence Chung et al. «Non-Functional Requirements in Software Engineering». Em: *International Series in Software Engineering*. 2000.

[41] COEPD Contributors. *What is FURPS+?* 2014. url: `https://businessanalysttraininghyderabad.wordpress.com/2014/08/05/what-is-furps/`.

[42] Ana Figuiredo. *SINFE - Análise e Design*. 2020. url: https://moodle.isep.ipp.pt/pluginfile.php/205179/mod_resource/content/1/Revis%C3%A3o_UML_SINFE_2019_2020.pdf.

[43] James Turnbull. *The Art of Monitoring*. James Turnbull, 2014. isbn: 9780988820241. url: https://www.amazon.com/Art-Monitoring-James-Turnbull-ebook/dp/B01GU387MS.

[44] Simon Brown. *The C4 model for Visualising Software Architecture*. url: https://c4model.com/.

[45] Philippe Kruchten. «The 4+1 View Model of architecture». Em: *IEEE Software* 12.6 (nov. de 1995), pp. 42–50. doi: 10.1109/52.469759.

[46] Finereport. *3-tier architecture: Everything you need to know*. Acedido em Mar 2023. url: https://www.finereport.com/en/product-functions/3-tier-architecture.html.

[47] OpenTelemetry Contributors. *.NET Instrumentation for OpenTelemetry*. url: https://github.com/open-telemetry/opentelemetry-dotnet-contrib/tree/Instrumentation.AspNet-1.0.0-rc9.8.

[48] OpenTelemetry Team. *.NET Instrumentation for OpenTelemetry*. Acedido em Mai 2023. url: https://github.com/open-telemetry/opentelemetry-dotnet/tree/core-1.4.0.

[49] Daniel Khan et al. *The best (and worst) reasons to adopt OpenTelemetry*. Acedido em Abr 2023. url: https://devops.com/the-best-and-worst-reasons-to-adopt-opentelemetry/.

[50] OpenTelemetry Contributors. *OpenTelemetry Collector Contrib*. Acedido em Mai 2023. url: https://github.com/open-telemetry/opentelemetry-collector-contrib.

[51] Chao Wang e Xiaobing Li. *Fast and reliable schema-agnostic log analytics platform*. Fev. de 2021. url: https://www.uber.com/en-IN/blog/logging/.

[52] Ankit Anand. *Launching support for Clickhouse*. Jun. de 2021. url: https://signoz.io/blog/clickhouse-storage-monitoring/.

[53] Alexey Milovidov. *A benchmark for analytical DBMS*. Acedido em Mai 2023. url: https://benchmark.clickhouse.com/.

[54] Jesse Howarth. *Why discord is switching from Go to Rust*. Ago. de 2021. url: https://discord.com/blog/why-discord-is-switching-from-go-to-rust.

[55] ClickHouse Team. *Usage recommendations*. Out. de 2019. url: https://clickhouse.com/docs/ru/operations/tips.

[56] Konrad Beiske. *Zookeeper - the king of coordination*. Abr. de 2019. url: https://www.elastic.co/blog/found-zookeeper-king-of-coordination.

[57] ClickHouse Team. *Why is Clickhouse Keeper recommended over zookeeper?* Acedido em Jun 2023. url: https://clickhouse.com/docs/knowledgebase/why_recommend_clickhouse_keeper_over_zookeeper.

[58] Altinity Team. *Clickhouse-Keeper Stability*. Nov. de 2022. url: https://kb.altinity.com/altinity-kb-setup-and-maintenance/altinity-kb-zookeeper/clickhouse-keeper/.

[59] SigNoz Team. *Set Up Distributed ClickHouse for SigNoz*. Acedido em Abr 2023. url: https://signoz.io/docs/operate/clickhouse/distributed-clickhouse.

[60] Phillip Carter e OpenTelemetry Team. *OpenTelemetry Collector Specification Status*. Acedido em Abr 2023. url: https://github.com/open-telemetry/opentelemetry-collector.

Apêndice A

Status Monitor WatchDog

Este apêndice tem como propósito mostrar algumas imagens ilustrativas da interface gráfica da ferramenta de monitorização atualmente em uso na Ciberbit, o Status Monitor.

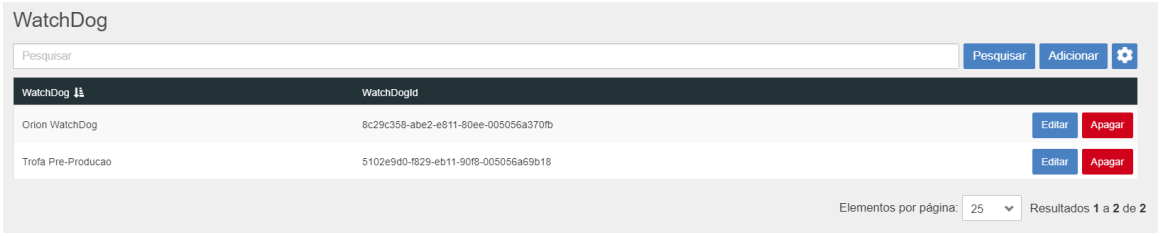

Figura A.1: Status Monitor - Listagem de WatchDogs.

Figura A.2: Status Monitor - Listagem Lista de Configurações de WatchDogs.

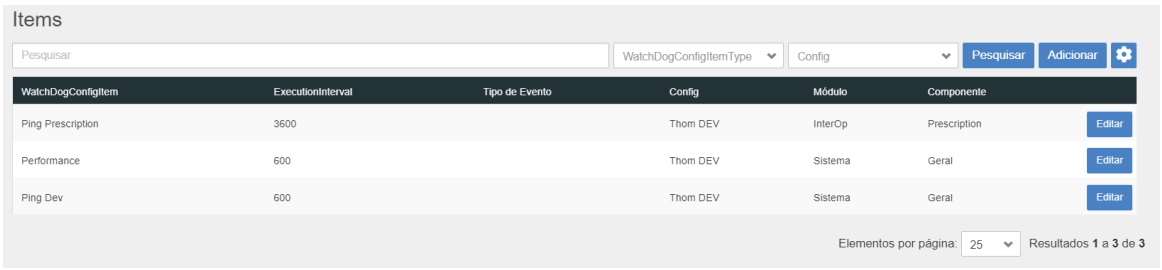

Figura A.3: Status Monitor - Listagem de Items de Configurações.

Apêndice B

Grau de cumprimento dos requisitos

Este apêndice apresenta um relatório que visa mostrar os requisitos cumpridos pela solução.

B.1 Requisitos funcionais

Como referido na secção 7.2, apenas não foi possível implementar corretamente a camada de instrumentação.

B.2 Atributos de qualidade

As tabelas seguintes referem os requisitos não funcionais, também designados por atributos de qualidade, do projeto. A numeração atribuída aos requisitos é a mesma presente na lista da subsecção 3.2.2.

B.2.1 Funcionalidade

Tabela B.1: Cumprimento dos atributos de funcionalidade.

N°	Progresso	Detalhes
R.NF.3	Feito	Via SigNoz Frontend.
R.NF.6	Feito	Via SigNoz AlertManager.

B.2.2 Usabilidade

Tabela B.2: Cumprimento dos atributos de usabilidade.

N°	Progresso	Detalhes
R.NF.5	Feito	Via SigNoz Frontend.

B.2.3 Confiabilidade

Tabela B.3: Cumprimento dos atributos de confiabilidade.

N°	Progresso	Detalhes
R.NF.7	Feito	Via HTTPS.
R.NF.8	Feito	Via HTTPS.
R.NF.15	Feito	Via Docker. A ferramenta docker-compose encontra-se configurada para levanar qualquer contentor que falhe.

B.2.4 Desempenho

Tabela B.4: Cumprimento dos atributos de desempenho.

N°	Progresso	Detalhes
R.NF.12	Feito	Via SigNoz Frontend.
R.NF.14	Feito	Testado no capítulo 6.

B.2.5 Suportabilidade

Tabela B.5: Cumprimento dos atributos de suportabilidade.

N°	Progresso	Detalhes
R.NF.1	Não Feito	Não foi implementada a camada de instrumentação.
R.NF.2	Feito	A *pipeline* foi projetada por forma a desacoplar a ferramenta de geração de dados da ferramenta de análise.
R.NF.9	Feito	Via SigNoz Frontend.
R.NF.10	Feito	A *pipeline* foi construída com recurso a ferramentas conhecidas por permitir uma escalabilidade horizontal.

B.2.6 Restrições de desenho

Nenhuma restrição foi proposta.

B.2.7 Restrições de implementação

Nenhuma restrição foi proposta.

B.2.8 Restrições de interface

Tabela B.6: Cumprimento das restrições de interface.

N°	Progresso	Detalhes
R.NF.4	Feito	Via NGINX e Let's Encrypt SSL.
R.NF.11	Feito	Via OpenTelemetry Collector.
R.NF.13	Feito	Via SigNoz AlertManager.

B.2.9 Restrições físicas

Nenhuma restrição foi proposta.