Fatec Americana Ministro Ralph Biasi   CPS Centro Paula Souza   SÃO PAULO GOVERNO DO ESTADO

Faculdade de Tecnologia de Americana "Ministro Ralph Biasi"

Avaliação de segurança e escalabilidade do Elixir em ambientes nuvem de IoT

Safety and scalability assessment of Elixir in cloud IoT environments

Henrique Nascimento de Carvalho, Fatec Americana
henrique.carvalho9@fatec.sp.gov.br

Marcus Vinícius Lahr Giraldi, Fatec Americana
marcus.lahr@fatec.sp.gov.br

Resumo

A crescente popularidade de dispositivos IoT exige arquiteturas de backend robustas e seguras. Este trabalho investiga como as linguagens Elixir e Erlang podem ser usadas para desenvolver sistemas em nuvem para IoT escaláveis e seguros. A metodologia inclui revisão bibliográfica e experimentos comparando desempenho, escalabilidade e eficiência energética de Elixir e Node.js. Os resultados sugerem que Elixir é vantajoso em escalabilidade e resiliência, enquanto Node.js se destaca em latência e processamento de requisições. Conclui-se, portanto, que a escolha entre Elixir e Node.js dependerá dos requisitos específicos de projeto.

Palavras-chave: Internet das coisas, Erlang, Elixir, Segurança em IoT, Escalabilidade

Abstract

The growing popularity of IoT devices demands robust and secure backend architectures. This work investigates how the Elixir and Erlang languages can be used to develop scalable and secure cloud-based systems for IoT. The methodology includes a literature review and experiments comparing the performance, scalability, and energy efficiency of Elixir and Node.js. The results suggest that Elixir is advantageous in scalability and resilience, while Node.js excels in latency and request processing. Therefore, it is concluded that the choice between Elixir and Node.js will depend on the specific project requirements.

***Keywords**: Internet of Things, Erlang, Elixir, Security in IoT, Scalability*

1. Introdução

A crescente popularidade dos dispositivos IoT (*Internet of Things*), traz à necessidade de arquiteturas de *backend* nuvem robustas e seguras. Neste contexto, exploramos o potencial das linguagens Elixir e Erlang, conhecidas por sua robustez em telecomunicações, para o desenvolvimento de sistemas IoT escaláveis e seguros.

De acordo com a IBM (2024), define-se um dispositivo IoT como qualquer objeto com sensores e conectividade para troca de dados com a internet, promovendo algum tipo de automação e interação. Os primeiros exemplos de dispositivos como este surgiram nos anos 80, de acordo com o site da CMU *School Of Computer Science* (2024), em 1982 uma máquina de venda de refrigerantes modificada da Coca-Cola na Universidade Carnegie Mellon tornou-se o primeiro utensílio conectado à ARPANET e era capaz de reportar seu próprio inventário e se as bebidas estavam em temperatura desejável.

Ressalva-se, porém, que a utilização e surgimento destes dispositivos implica maiores riscos de segurança em ambientes e redes que os utilizam, dado por conta do descaso na criação deles, priorizando a produção destes em massa sem levar em conta as possíveis vulnerabilidades contidas. De acordo com o relatório da OWASP de 2018, cita-se algumas vulnerabilidades comuns destes dispositivos como: Senhas fracas chumbadas em código e não configuráveis; Redes inseguras; Ecossistemas inseguros; Sistemas de atualização inseguros do dispositivo; Componentes defasados; Falta de proteções de privacidade de usuários; Transferências inseguras de dados; Administração imprópria do dispositivo e Configurações padrão inseguras.

Os benefícios principais em termos de segurança que justifiquem a utilização de uma linguagem mais robusta como o Erlang poderiam ser aproveitados no que cabe às senhas fracas chumbadas, ecossistemas ou sistemas de atualização inseguros e a transferência insegura de dados. De acordo com a própria página principal da linguagem, o Erlang é focado em telecomunicações, criado pela Ericcson

em 1986. Por conta disso foi feita voltada para escalabilidade eficiente, concorrência robusta e tolerância a falhas, para ocorrer processamento rápido e confiável de dados em tempo real.

De acordo com o documentário da Honeypot (2018) sobre a linguagem, em 2011, José Valim em um Projeto de R&D da Plataformatec, uma empresa de consultoria de software brasileira, criou o Elixir, uma linguagem baseada no Erlang que, quando compilada e executada na mesma máquina virtual da mesma, a BEAM (Máquina Abstrata Erlang). O objetivo é modernizar o Erlang, que conta com uma sintaxe mais complexa e de difícil leitura, visando facilitar a manutenibilidade de código via recursos de tipagem, orientação a objeto e sintaxe mais verbosa.

Em suma, as características intrínsecas de Elixir e Erlang, como tolerância a falhas e escalabilidade eficiente, podem ser aproveitadas na infraestrutura da nuvem para mitigar esses problemas e melhorar a segurança e a confiabilidade dos sistemas IoT quando lidamos com a nuvem, porém atualmente não há muita utilização destas para este caso. Cabendo então a exploração da eficácia destes.

2. Potencial do Elixir em Arquiteturas para IoT

Há diversas discussões no que cabem a qual arquitetura de software é a mais adequada, além também da linguagem mais eficaz para implementação dele. Em Fedrecheski et. al (2016), é explorada a utilização de linguagens de programação, como Erlang e Elixir, para implementações dentro de um contexto arquitetural *Swarm*, definido como um ambiente de comunicação máquina-a-máquina com foco em dispositivos diversos e orientados por máquinas.

2.1. Definição e uso do Elixir

A pesquisa realizada pelos autores em Fedrecheski et. al (2016) conduz uma análise comparativa entre as implementações de um *Swarm Broker* em Java e Elixir, com ênfase em aspectos como tempo de resposta de solicitações HTTP, quantidade de linhas de código, utilização de CPU e memória. Os resultados indicam que o Elixir demonstra vantagens na quantidade de linhas de código e no uso de memória, enquanto o Java consome ligeiramente menos CPU. No entanto, o Java enfrenta desafios em relação ao tempo de resposta em situações de alta demanda. Os autores sugerem, portanto, que o Elixir possui um potencial promissor para a execução eficiente de micro serviços na nuvem, e pode desempenhar um papel fundamental na Internet das Coisas.

Outro estudo, realizado por Haenisch (2016) abordou código escrito em duas linguagens para uma estação base, que lidava principalmente com tratamento de erros e funções de monitoramento. A versão original do código consistia em 620 linhas de código-fonte, uma mistura de C, Ruby e Python. No entanto, após a reavaliação dos requisitos do usuário, o tamanho do código foi reduzido para 251 linhas de Ruby, e uma reimplantação em Elixir resultou em apenas 106 linhas de código.

O autor também comparou a complexidade das implementações e descobriu que a versão em Elixir não apenas era muito menor em tamanho, mas também tinha uma complexidade significativamente menor em comparação com a versão em Ruby. E concluiu que o código em Elixir provavelmente possui menos erros de software.

Já em Jaisinghani et al. (2018), foram também realizados estudos sobre a transmissão por nós de IoT em redes densas baseadas em *WLANs*. Foi implementado um protótipo utilizando a linguagem Elixir, o estudo concluiu que a abordagem utilizando o Elixir reduziu o consumo de energia em 18% para os dispositivos IoT, argumentando que ele também demonstrou ser eficaz na transmissão de dados em condições de rede desafiadoras e apresentou e eficiência de transmissão em

comparação com abordagens tradicionais de associação *WiFi*.

2.2. Requisitos de performance em IoT

De acordo com Buyya; Dastjerdi (2016) na avaliação de linguagens de programação para micro serviços utilizados por serviços IoT na nuvem, diversos recursos relevantes entram em jogo. A escalabilidade é um aspecto crucial, pois os sistemas IoT requerem suporte a padrões de programação diversos para lidar dinamicamente com o balanceamento de carga. A concorrência é de extrema importância, já que a comunicação em tempo real entre milhões de dispositivos exige uma gestão eficiente das conexões simultâneas, tornando o bloqueio de threads menos prático.

Conforme descrito em Padmanabhan; Arjun (2020) A capacidade de coordenação em uma linguagem de programação é fundamental, quer seja de forma explícita (orientada por controle) ou implícita (orientada por dados) na orquestração do papel dos elementos computacionais no ecossistema IoT. O suporte à heterogeneidade é outro fator-chave, garantindo que o framework de programação fornece orientações sobre como mapear cálculos para diversos elementos computacionais.

A tolerância a falhas é fundamental, uma vez que as aplicações IoT precisam fazer transições suaves de estados *online* para *offline* quando ocorrem partições na rede. Uma transição leve, tanto em termos de sobrecarga de tempo de execução quanto de esforço de programação, é desejada para um desenvolvimento eficiente de sistemas. Além disso, o suporte à latência e sensibilidade é essencial, especialmente em aplicações geograficamente distribuídas, onde enviar todos os cálculos para a nuvem pode não ser ideal. O framework de programação escolhido deve lidar dinamicamente com esses requisitos para facilitar o desenvolvimento eficaz de IoT.

Portanto, os estudos enfatizam a importância da escolha da linguagem de programação no desenvolvimento de sistemas vinculados à nuvem de dispositivos IoT. Os estudos experimentais e análises comparativas sugeriram que o Elixir pode fornecer benefícios significativos em termos de eficiência de codificação, uso de memória e desempenho em comparação com outras linguagens de programação, como Java, Ruby e Python. No geral, estes estudos apoiam o argumento de que a linguagem de programação pode desempenhar um papel crucial na criação de sistemas eficazes e eficientes. No entanto, o Elixir é pouco utilizado neste campo, conforme mencionado por Nishiguchi (2021), C e Java ainda são amplamente utilizados no desenvolvimento desses sistemas.

3. Metodologia

Em termos metodológicos, o projeto visa realizar uma avaliação da linguagem Elixir, realizando um comparativo dela com Node.js, para o desenvolvimento de *backend* em nuvem para IoT. O projeto inicialmente terá uma revisão bibliográfica, na qual serão identificados e analisados os fundamentos de ambas as linguagens assim como a importância do desenvolvimento de software com qualidade para este fim. Isso envolverá a busca, seleção e análise crítica de diversas fontes teóricas, que podem incluir: Livros e monografias artigos sobre IoT e estas linguagens de programação; Teses e dissertações que abordam o uso de Elixir em projetos IoT; Artigos científicos em periódicos especializados; Documentação técnica, tutoriais e guias de implementação relacionados ao Elixir e Node.js

A revisão bibliográfica permitirá estabelecer um conhecimento sólido sobre os fundamentos teóricos e as melhores práticas neste meio, assim como os desafios comuns que são encontrados, que podem ser replicadas em testes com as linguagens.

Além da revisão bibliográfica, o projeto também incluirá a realização de experimentos práticos em Elixir e Node com o objetivo de avaliar o desempenho e a eficácia destas no contexto dos da nuvem para sistemas IoT. Esses experimentos podem incluir: Desenvolvimento de protótipos de sistemas IoT simulados usando Elixir e Node, junto de Avaliação de métricas de desempenho, escalabilidade e eficiência energética. De maneira que os experimentos práticos forneçam informações empíricas sobre a capacidade do Elixir em atender às demandas práticas de implementações comumente utilizadas em nuvem.

Após a conclusão da revisão bibliográfica e dos experimentos práticos, os resultados serão analisados e sintetizados. A análise comparará as informações teóricas obtidas na revisão bibliográfica com as descobertas práticas dos experimentos em Elixir.

4. Importância do software em segurança de dados

Conforme dito em Pfleeger, C; Pfleeger, S; Coles-Kemp (2024) e Kidd (2020) quando trata-se de segurança da informação, lida-se com o conceito de confidencialidade, integridade e disponibilidade. A confidencialidade cabe a garantia de que somente quem deveria ter acesso a dados específicos de fato o tenha, em um ambiente nuvem que receba dados de algum dispositivo, isso é garantido através de considerações de segurança dos *endpoints* que deveriam ter acesso a este; A integridade, porém, lida com a consistência e precisão dos dados em todo seu ciclo de vida, quando lidamos com software isso implica na garantia de que dados recebidos estejam sendo processados adequadamente; A disponibilidade, quando tratamos de software, refere-se a capacidade de um sistema em se recuperar rapidamente e que recursos deste permaneçam acessíveis pelos clientes que dependam deste. Considera-se daí estes três conceitos separadamente, relacionando-os com considerações importantes na arquitetura de software quando tratamos de ambientes de nuvem em IoT.

Como mencionado anteriormente, da confidencialidade, implica na necessidade de controle de acesso de usuários e/ou outros sistemas. Neste caso a importância da implementação de software se dá na medida em que tratamos do gerenciamento de acessos e criptografia de dados, deste trata-se da necessidade de criptografar dados sensíveis que sejam salvos localmente no serviço em nuvem, garantindo que mesmo que expostos estes dados não são facilmente acessíveis sem as chaves apropriadas, já daquele há a consideração de autenticações multifatoriais que garantam que somente aqueles que de fato devem, podem acessar o sistema em nuvem e processar dados neste.

Já em integridade, a complexidade é maior, pois para garantir que os dados estejam de fato íntegros na nuvem, é preciso que o serviço processe os adequadamente, sem que sejam salvos dados incorretos ou indevidos. Tratamos aqui da validação e verificação de dados, transferência e consistência dos mesmos e logs de auditoria disponíveis. O primeiro considera a utilização de versionamento de cargas recebidas em nuvem e da validação dos dados que serão processados; O segundo garante que dados salvos sejam consistentes principalmente quando lidamos com banco de dados, cabe aqui muitas vezes a implementação de bloqueio otimista ou pessimista de escrita em banco; O terceiro visa facilitar na investigação de problemas, salvados dados que podem ser consultados para investigar incidentes.

Em termos de disponibilidade, a estratégia visa assegurar acesso contínuo e eficaz aos dados e serviços na nuvem. Isso envolve implementar redundância de sistemas e dados, escalabilidade automática de recursos e monitoramento proativo. A redundância, tanto em hardware quanto em software, garante que existam recursos de backup prontos para assumir em caso de falhas, mantendo o serviço sempre ativo. A escalabilidade permite ajustar os recursos de forma dinâmica conforme a demanda, evitando sobrecarga e mantendo o desempenho estável. Por fim, o monitoramento proativo identifica e resolve problemas emergentes rapidamente, minimizando

potenciais interrupções e garantindo que os serviços permaneçam disponíveis e operacionais para os usuários finais. Essas medidas são cruciais para suportar operações contínuas e confiáveis, especialmente em ambientes de alta demanda.

A importância dos pilares de confidencialidade, integridade e disponibilidade no desenvolvimento de software para ambientes em nuvem de IoT é indiscutível. À medida que exploramos a eficácia de Elixir para atender a esses requisitos, surgem perguntas críticas sobre como essa linguagem se compara a outras opções populares, como Node.js, especialmente em termos de segurança e escalabilidade.

5. Comparação de linguagens: Elixir e Node.js

Para fim deste estudo, toma-se a utilização do Node.js já que ele é robusto para desenvolvimento backend em serviços de nuvem devido à sua arquitetura assíncrona e não bloqueante, que otimiza o desempenho ao lidar com múltiplas solicitações simultâneas, essencial para aplicações escaláveis na nuvem. Além disso, a popularidade do Node.js, conforme evidenciado pela Pesquisa de Desenvolvedores do Stack Overflow de 2023 como menciona Doerrfeld (2023), mostra que ele é frequentemente utilizado para aplicações web backend, refletindo sua eficácia e aceitação na comunidade de desenvolvedores. A ampla adoção facilita a utilização do mesmo em ambiente produtivo na medida que há várias bibliotecas disponíveis, tornando-o ideal para ambientes dinâmicos de nuvem.

Além disso, como mencionado em Borbély (2023), o Typescript otimiza a manutenibilidade do Node nas organizações, na medida em que adiciona uma camada de tipagem ao Javascript, mesmo que não seja necessariamente utilizado em tempo de execução, ao menos pode facilitar no entendimento do código por parte de outros desenvolvedores e impedir que certas validações sejam relevadas e causem problemas futuros.

Já o Elixir, explicitado na Elixir Wiki (2024), baseado na máquina virtual Erlang,

é desenhado para lidar com alta concorrência, falhas no sistema e responder rapidamente em ambientes distribuídos. Essas características podem ser incrivelmente úteis para manter a disponibilidade e a integridade de aplicações IoT. Mas isso nos leva a questionar se o Elixir realmente oferece vantagens significativas em comparação com o Node.js, que como indicado pela Openjs Foundation (2024) na área de aprendizado, é conhecido por sua habilidade em gerenciar I/O assíncrono e operações em tempo real.

Por outro lado, enquanto o Elixir tem seus pontos fortes na gestão de estado em sistemas distribuídos, o Node.js tem uma comunidade maior e uma ampla variedade de bibliotecas, o que pode tornar mais fácil implementar medidas de segurança fortes e eficazes. Isso nos deixa com algumas perguntas cruciais para pesquisa quando levando em consideração a segurança da informação.

Confidencialidade: Como as funcionalidades de segurança do Elixir, especialmente em termos de criptografia e gerenciamento de sessões, se comparam com as do Node.js em ambientes de produção críticos?

Integridade: Será que o Elixir é melhor em garantir a integridade dos dados em situações de alta concorrência, graças à sua imutabilidade e ao modelo de atores, comparado ao Node.js?

Disponibilidade: As características de tolerância a falhas do Elixir resultam em uma melhor disponibilidade e resiliência em comparação com o Node.js, especialmente durante picos de demanda ou falhas de componentes?

6. Flexibilidade de bibliotecas

Como apontado por Martin (2008), ter uma variedade de bibliotecas disponíveis para uma linguagem de programação em ambientes produtivos é crucial porque amplia significativamente as capacidades e a flexibilidade dos desenvolvedores. Isso permite que eles implementem soluções mais rapidamente, utilizando bibliotecas testadas e comprovadas, ao invés de terem que desenvolver funcionalidades complexas do zero. Além disso, a diversidade de bibliotecas ajuda a garantir que uma linguagem permaneça relevante e adaptável às novas tendências e demandas tecnológicas, aumentando a eficiência e a inovação.

6.1. Bibliotecas do Elixir

O repositório *awesome-elixir* criado por Beckmann (2024) compila várias bibliotecas úteis para o desenvolvimento em Elixir, abrangendo uma ampla gama de funcionalidades, recebendo contribuições e atualização constante, para fins deste estudo toma como base algumas das bibliotecas recomendadas deste repositório.

No que cabe ao atendimento de requisitos de confidencialidade, considera-se aqui a disponibilidade de bibliotecas que tratem da autenticação de usuários e segurança na persistência destes dados. Da lista compilada, podem-se mencionar algumas bibliotecas como: Comeonin, que suporta várias funções de hash de senha, como BCrypt, Argon2 e Pbkdf2, que são essenciais para armazenar senhas de forma segura; há também a Guardian, que possibilita a utilização de JSON Web Tokens (JWT), um padrão de token compacto e seguro para troca de informações entre partes, usados frequentemente para autenticação e troca de informações entre cliente e servidor, estas informações foram retiradas das páginas principais das bibliotecas citado em Whitlock (2024) e Ueberauth (2024).

Ademais, levando em consideração a integridade, vale-se a utilização de bibliotecas que auxiliam no gerenciamento de dados. Aqui pode-se mencionar: *Ecto*,

como citado em Elixir-ecto (2024) trata-se de um *wrapper* de banco de dados que facilita na migração de base de dados, o conceito de *wrapper* cabe a uma estrutura que embrulha ou encapsula uma funcionalidade, desta forma garante maior segurança na alteração de esquemas de banco e a *ExAudit*, que como mencionada em Zenner IoT Solutions (2024), utiliza da biblioteca *Ecto* como base para fornecer funcionalidades de auditoria e versionamento de dados.

A garantia de disponibilidade, porém, vem da adoção de bibliotecas que auxiliam na tolerância a falhas de sistema. Aqui menciona-se: *fuse*, criado por Andersen (2024), um quebra-circuitos para Elixir, este conceito em programação cabe na capacidade de travar o sistema para que falhas em cascata não aconteçam, permitindo que sejam retomadas depois de um período de recuperação ou manutenção específico e *LibCluster*, criado e descrito por Schoenfelder (2024) que possibilita a criação de nós distribuídos, o que permite maior escalabilidade do sistema, permitindo que diversos nós possam continuar operando caso um destes falhe.

Porém, como destacado pela Elixir Wiki (2024), vale considerar que Elixir em si é construída em cima da máquina virtual Erlang, que é conhecida por suas capacidades de sistemas distribuídos e tolerância a falhas. Isso significa que muitos dos requisitos para disponibilidade e resiliência são intrínsecos à plataforma. E, portanto, podem ser vantajosos mesmo que desprovidos de bibliotecas específicas.

6.2. Bibliotecas do Node.js

De acordo com Openjs Foundation (2024) o ecossistema Node.js é rico em bibliotecas que facilitam o desenvolvimento de aplicações robustas, oferecendo uma variedade de ferramentas para atender a diversos requisitos de programação. Especificamente, para atender aos pilares de confidencialidade, integridade e disponibilidade, existem várias bibliotecas notáveis.

Quanto à confidencialidade, o Node.js possui uma série de bibliotecas que ajudam na autenticação de usuários e na segurança da persistência de dados. Dentre elas, destacam-se: *bcrypt.js*, criada e descrita por Keletiv (2024), que suporta funções de *hash* de senha robustas, como *BCrypt*, crucial para o armazenamento seguro de senhas e *jsonwebtoken*, criada pela Auth0 (2024), que permite a geração e verificação de *JSON Web Tokens*.

Em relação à integridade, o Node.js oferece ferramentas que auxiliam no gerenciamento eficiente de dados: *crypto*, um módulo nativo, criado pela Openjs Foundation e descrito em Crypto (2024), que fornece funcionalidades criptográficas completas, garantindo a segurança e a integridade dos dados através de funções de *hash*, *HMAC* e outros e *joi*, que como mencionado por Hapi.js (2024) trata-se de uma biblioteca para validação de dados que assegura que as entradas do usuário ou de outras fontes sigam um esquema específico, prevenindo a inserção de dados incorretos ou maliciosos no sistema.

Por fim, a disponibilidade é assegurada por meio de bibliotecas que promovem a tolerância a falhas e a resiliência do sistema: *pm2*, descrito por Strzelewicz (2024) é um gerenciador de processos avançado para Node.js, capaz de manter a aplicação funcionando continuamente, reiniciando-a automaticamente em caso de falha e *nodemon*, que criado pela Remy (2024), embora comumente usados em desenvolvimento, monitora alterações nos arquivos do projeto e reinicia automaticamente o servidor, facilitando a manutenção contínua e a disponibilidade durante o desenvolvimento.

Assim como o Elixir, o próprio ambiente de execução do Node também garante alguns dos pilares, como descrito pela Openjs Foundation (2024) este é baseado no motor V8 do Google e é otimizado para performance e eficiência, contribuindo significativamente para a disponibilidade e o desempenho de aplicações em tempo real.

6.3. Estatísticas de uso e manutenibilidade de bibliotecas

Na página de aprendizado da Openjs Foundation (2024) menciona-se que em Setembro de 2022 o gerenciador de pacotes NPM, que inclui grande parte dos pacotes públicos disponíveis para Javascript, detinha de mais de 2.1 milhões de bibliotecas, mesmo que nem todas estas sejam mantidas ou até mesmo com tipagem disponível para serem utilizadas com Typescript, isso demonstra ainda assim um número consideravelmente grande de implementações e algoritmos para resolver diversos problemas específicos que certos requisitos devam atender.

Para fins de demonstrar a manutenibilidade e usabilidade de pacotes disponíveis para Node.js considera-se um ranking de bibliotecas criado por Kashcha (2024), baseado no maior número de dependências que um pacote tenha de outros pacotes, visualizados no Quadro 1 com informações relevantes sobre o número de downloads, a última atualização do pacote e a quantidade de desenvolvedores contribuindo para o desenvolvimento dele.

No que cabe às bibliotecas mais populares disponíveis para Elixir, coleta-se os mesmos dados para fins de comparação, disponíveis no Quadro 2. A lista de bibliotecas mais populares foi coletada de uma página disponível na Elixir Wiki (2023), alguns dados não estavam disponíveis no gerenciador de pacotes Hex (2024), pois a biblioteca foi integrada à linguagem.

Quadro 1 – Dados coletados de bibliotecas em Node.js.

Nome	Downloads semanais	Contribuidores	Última atualização
lodash	45.949.005	289	3 anos atrás
chalk	273.219.777	57	10 meses atrás
request	12.406.215	310	4 anos atrás
commander	129.254.402	183	3 meses atrás
react	22.198.959	1.664	14 horas atrás
express	26.971.566	308	1 mês atrás
debug	224.191.286	113	2 anos atrás
async	46.797.657	236	6 meses atrás
fs-extra	84.862.921	92	5 meses atrás
moment	18.242.801	603	4 meses atrás

Fonte: Elaboração Própria a partir de dados do Github (2024) e NPM (2024).

Porém, cabe-se considerar que grande parte destas bibliotecas mencionadas na Wiki como mais populares foram desenvolvidas pelos fundadores da linguagem e, portanto, acabam por não serem totalmente coesas para uma comparação com bibliotecas do Node, que foram em sua grande parte criadas pela comunidade, fora do escopo interno da linguagem.

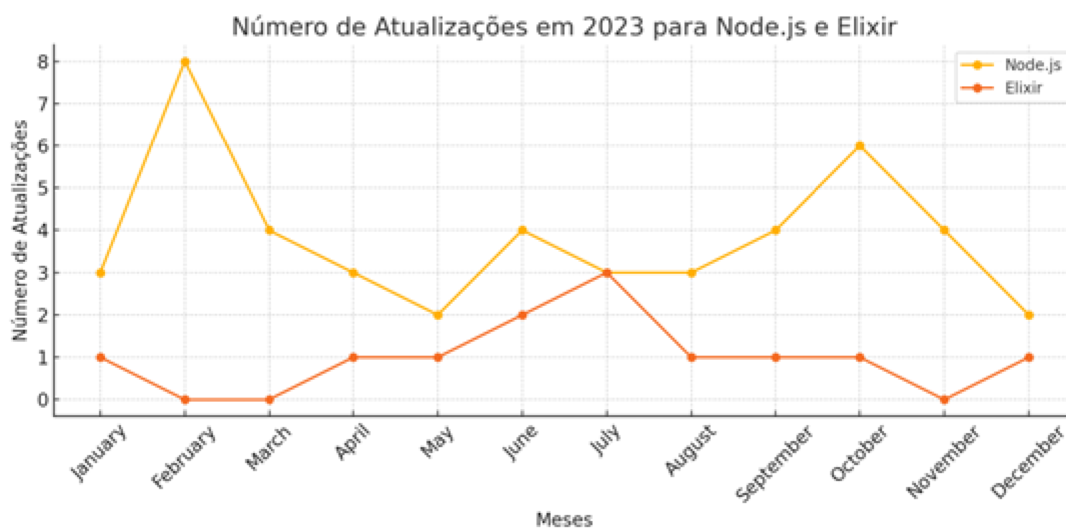
Por conta disso, analisa-se também o ritmo de atualizações de cada tecnologia, baseado nos seus repositórios do github e a quantidade de releases feitas por mês no período de 2023, os dados coletados estão demonstrados na Figura 1.

Quadro 2 – Dados coletados de bibliotecas em Elixir.

Nome	Downloads semanais	Contribuidores	Última atualização
GenServer	-	-	1 semana atrás
Ecto	258.203	741	2 meses atrás
Phoenix	249.632	1182	1 mês atrás
ExUnit	-	-	-
Plug	294.348	297	4 meses atrás
Poison	168.557	32	3 anos atrás
Timex	165.442	191	1 ano atrás
ExDoc	137.277	197	1 mês atrás
Credo	218.037	221	2 meses atrás
Distillery	18.032	158	5 anos atrás

Fonte: Elaboração Própria a partir de dados do Github (2024) e Hex (2024).

Figura 1 – Comparativo de atualizações Node e Elixir



Fonte: Elaboração Própria a partir de dados do Github (2024)

O Node possui mais atualizações mensais do que o Elixir em vista de sua popularidade e o fato que ele tenta manter compatibilidade com versões anteriores, portanto criando versões para outras principais anteriores a atual. Em termos de colaboradores, o repositório node (2024) possui atualmente cerca de 3.400 colaboradores no Github (2024), superior aos 1.300 do Elixir-lang (2024).

Os dados coletados destacam a diferença de tamanho entre as duas comunidades. Em um contexto corporativo, isso pode tornar a adoção de Elixir percebida como mais arriscada, devido ao menor suporte e à escassez de desenvolvedores disponíveis. No entanto, é importante lembrar que esses não são os únicos fatores determinantes para a escolha de uma linguagem. O tamanho da comunidade pode crescer com o aumento do interesse pela linguagem. Além disso, algumas oferecem soluções altamente especializadas, o que pode justificar sua escolha para determinados projetos ou necessidades específicas.

6.4. Legibilidade e acessibilidade de código

Ao comparar a sintaxe e clareza entre Node.js e Elixir, observam-se diferenças significativas que impactam diretamente a legibilidade e a manutenção destes. Como citado na documentação provida pela Openjs Foundation (2024) o Node.js, utilizando JavaScript, oferece uma ampla flexibilidade que resulta em uma sintaxe variada, de concisa a verbosa, dependendo das práticas do desenvolvedor. Essa flexibilidade pode introduzir complexidades, especialmente com o uso frequente de *callbacks* e manipulação de tipos, quando tratamos de Typescript. Além disso, a linguagem permite outras práticas estilísticas que variavam entre desenvolvedores, o que pode comprometer a clareza do código.

Já o Elixir, descrito pela The Elixir Team (2024) é reconhecido por sua sintaxe que embora não seja usualmente adotada, por ser funcional, quando adotada pode ser mais limpa e organizada, favorecendo um estilo de codificação alinhado. De uma

certa maneira, isso implica que embora a sintaxe não seja usual, ela ao menos é mais concisa que o Javascript pois não permite tanta estilização. A flexibilidade do Node.js pode resultar em inconsistências estilísticas entre diferentes projetos, tornando essencial o uso de ferramentas como *ESLint* para garantir uniformidade. Em contraste, Elixir promove uma consistência notável através de convenções sólidas e práticas suportadas tanto pela comunidade quanto pela linguagem em si, complementadas por ferramentas de análise estática e um compilador que asseguram a estrutura clara e ordenada do código, diferindo do Node como uma linguagem interpretada, e, portanto, não compilada. Essa consistência oferece vantagens significativas para o aprendizado de novos desenvolvedores, apesar de Elixir apresentar uma curva de aprendizado mais acentuada devido ao seu paradigma funcional.

Também vale salientar que mesmo dado de maior complexidade do que o Node, o Elixir ainda assim detém de uma linguagem de melhor entendimento do que o Erlang, como pode ser observado na Figura 2 e Figura 3, Elixir oferece uma sintaxe mais simples e direta, tornando-o potencialmente mais acessível. Em comparação, o Erlang requer uma estrutura mais formal e uma compreensão maior dos conceitos subjacentes da linguagem.

Ambas as linguagens são suportadas por comunidades ativas que priorizam documentação de alta qualidade e código bem comentado, observado na documentação de ambos os projetos, fatores que influenciam positivamente tanto a legibilidade quanto a facilidade de aprendizado.

Figura 2 – *Hello world* com Erlang

```erlang
-module(hello).
-export([greet/1]).

greet(Name) ->
    io:format("Hello, ~s!~n", [Name]).
```

Fonte: Elaboração própria

Figura 3 – *Hello world* com Elixir

```elixir
defmodule Hello do
  def greet(name) do
    IO.puts("Hello, #{name}!")
  end
end
```

Fonte: Elaboração própria

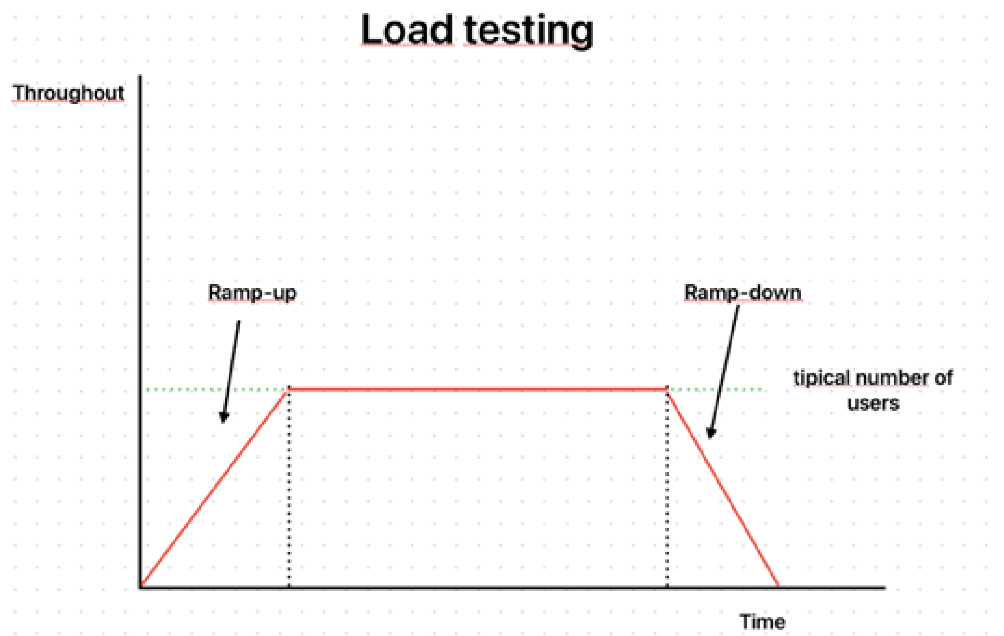
7. Experimentos

Para avaliar o desempenho em alguns cenários específicos, foram implementadas aplicações escritas em Elixir e Node.js, nas versões 1.16.3 e 20.14.0 respectivamente. O K6, criado e documentado pela Grafana Labs (2024), foi utilizado para implementação de testes de performance, trata-se de uma ferramenta de teste de carga e desempenho de código aberto. As aplicações foram configuradas para simular cenários reais de uso, onde múltiplos usuários acessam apis REST. A aplicação Node.js foi desenvolvida com o Expressjs (2024), enquanto a aplicação Elixir foi implementada com o Phoenix Framework (2024), ambas bibliotecas provêm simplificações para a exposição de APIs REST.

O K6 foi escolhido por sua capacidade de gerar testes de carga de maneira eficiente e por ser implementado em Go, uma linguagem que oferece excelente performance e suporte a *multi threading*, no caso do K6 isso permite que múltiplos usuários em nós distintos mandem dezenas de requisições ao mesmo tempo. O que torna o K6 uma ferramenta imparcial, capaz de testar qualquer backend de maneira neutra, independentemente da tecnologia subjacente.

Porém vale salientar que o K6 permite que os scripts de teste sejam escritos em JavaScript, isso não confere uma vantagem às aplicações Node.js. O K6 simula cargas de usuários e mede tempos de resposta, falhas de requisição e outras métricas de desempenho de maneira agnóstica, já que embora o código seja implementado em Javascript somente a lógica do script é utilizada, assim, o sistema executa estes separadamente utilizando o Go. Portanto, os resultados obtidos refletem a eficiência real das aplicações sob teste. Neste caso todos os scripts foram executados utilizando de uma estratégia de testes de carga simples, descrito por Minetto (2024), onde inicialmente a carga de usuários e requisições é baixa, aumenta e estabiliza por um determinado tempo e é diminuída gradativamente até o fim do teste, como demonstrado na Figura 4, demonstra-se também na Figura 5 um exemplo de resultado do K6.

Figura 4 – Demonstração da rampa em um teste de carga



Fonte: Retirado da página "Tipos de teste de carga" elaborado por Minetto (2024)

Figura 5 – Demonstração do resultado de um teste do K6

```
✓status is 200
✓data1 is correct
✓data2 is correct
✓data3 is correct

checks.........................: 100.00% ✓177800     ✗0
data_received..................: 21 MB   35 kB/s
data_sent......................: 4.0 MB  6.6 kB/s
✓ errors.........................: 0       0/s
http_req_blocked...............: avg=1.02µs   min=0s      med=0s      max=3.99ms p(90)=0s       p(95)=0s
http_req_connecting............: avg=137ns    min=0s      med=0s      max=1ms    p(90)=0s       p(95)=0s
✓ http_req_duration..............: avg=41.99ms  min=28.05ms med=40.18ms max=1.18s  p(90)=50.74ms  p(95)=54.84ms
  { expected_response:true }...: avg=41.99ms  min=28.05ms med=40.18ms max=1.18s  p(90)=50.74ms  p(95)=54.84ms
http_req_failed................: 0.00%   ✓0          ✗44450
http_req_receiving.............: avg=193.78µs min=0s      med=0s      max=3.83ms p(90)=759.51µs p(95)=916.7µs
http_req_sending...............: avg=2.83µs   min=0s      med=0s      max=1ms    p(90)=0s       p(95)=0s
http_req_tls_handshaking.......: avg=0s       min=0s      med=0s      max=0s     p(90)=0s       p(95)=0s
http_req_waiting...............: avg=41.79ms  min=28.05ms med=40ms    max=1.18s  p(90)=50.52ms  p(95)=54.67ms
http_reqs......................: 44450   73.956831/s
iteration_duration.............: avg=1.04s    min=1.02s   med=1.04s   max=2.18s  p(90)=1.05s    p(95)=1.06s
iterations.....................: 44450   73.956831/s
response_times.................: avg=41.99ms  min=28.05ms med=40.18ms max=1.18s  p(90)=50.74ms  p(95)=54.84ms
vus............................: 1       min=1       max=100
vus_max........................: 100     min=100     max=100
```

Fonte: Teste executado localmente para aplicação em Elixir

7.1. Experimentos de escala

Foi realizado um pequeno experimento de escala utilizando rampas no K6, aumentando a quantidade de usuários em cinco estágios, começando com 50 usuários virtuais até 200 usuários virtuais como mostrado na Figura 4. De maneira que o teste pretende demonstrar como cada aplicação escrita em sua tecnologia específica se comportaria conforme a quantidade de usuários aumenta. Com base nos resultados obtidos nos testes, é possível observar algumas diferenças e semelhanças significativas entre as implementações em Node.js e Elixir.

O Node.js demonstrou um desempenho superior em termos de latência, com um tempo de resposta p95 menor que o Elixir como observado no Quadro 3. Node.js também completou um número ligeiramente maior de requisições e testes, sugerindo uma capacidade de processamento marginalmente melhor sob carga.

Por outro lado, Elixir, apesar de ter um tempo de resposta p95 mais alto, ainda conseguiu um número de requisições e testes próximo ao de Node.js. Isso o torna uma escolha viável, especialmente em sistemas que podem tolerar uma latência ligeiramente maior em troca de benefícios como maior resiliência e escalabilidade, características inerentes à plataforma BEAM. Portanto, enquanto Node.js claramente possuiu uma vantagem em termos de latência, Elixir ainda assim foi altamente competitivo.

Quadro 3 – Resultados coletados de testes de escalabilidade usando o K6

Linguagem	Métrica	Valor	Unidade
Node.js	Requisições	168.045	Soma
	p (95) Tempo de resposta	4,5	Milissegundos
	Dados recebidos	43	Megabytes
	Testes	336.090	Soma
Elixir	Requisições	166.001	Soma
	p (95) Tempo de resposta	18,02	Milissegundos
	Dados recebidos	42	Megabytes
	Testes	332.002	Soma

Fonte: Elaboração Própria a partir de k6 e código local

7.2. Experimentos de assincronicidade

Para testar assincronicidade a ideia foi criar um *endpoint* em Elixir e Node que faça algumas requisições assíncronas para uma API externa, buscando analisar se há alguma grande diferença de desempenho. Foi criado um script utilizando o K6 para este teste, utilizando também a estratégia de rampa como na Figura 4. Os resultados também foram semelhantes, demonstrando que ambas as linguagens têm capacidade de realizar essas operações.

Aqui, como demonstrado no Quadro 4, os resultados também foram relativamente próximos em termos de requisições totais, número de testes executados e dados recebidos, de maneira que o Node é capaz de executar uma quantidade um pouco maior de requisições. O que difere e pode acarretar consequências maiores na escolha da tecnologia, porém, é o tempo de execução ligeiramente maior para o Elixir,

que talvez seja negligenciável dependendo do contexto.

É importante considerar que como mencionado na Elixir Wiki (2024) o Elixir usa uma máquina virtual que é bem competente na gestão de diversos processos, esta tem um custo operacional maior quando se trata da gestão destes processos, o que pode contribuir para durações de requisição ligeiramente mais altas. Em contrapartida, como dito pela Openjs Foundation (2024) o Node.js utiliza um modelo de I/O não bloqueador e orientado a eventos, de maneira que a execução não é gerenciada e é *single-threaded*, portanto não é necessário coordenador as operações. Neste caso específico de uso onde foi feito requisições HTTP assíncronas, Node.js é muito eficiente e opera a um custo mínimo, levando a um desempenho ligeiramente melhor em termos de duração das requisições, porém em cenários mais especializados a capacidade da máquina virtual do Erlang de operar com *multi-thread* talvez seja vantajoso.

Quadro 4 – Resultados coletados de testes de assincronicidade usando o K6

Linguagem	Métrica	Valor	Unidade
Node.js	Requisições	45.146	Soma
	p (95) Tempo de resposta	35,86	Milissegundos
	Dados recebidos	22	Megabytes
	Testes	180.584	Soma
Elixir	Requisições	44.450	Soma
	p (95) Tempo de resposta	54,84	Milissegundos
	Dados recebidos	21	Megabytes
	Testes	177.800	Soma

Fonte: Elaboração Própria a partir da execução do K6 com código local

7.3. Experimento de tolerância a falhas

Neste experimento foi criado dois endpoints na aplicação, um *api/health* para validar a saúde da aplicação, ou seja, se ela ainda está rodando e outro *api/crash* um *endpoint* destinado a criar um sinal de finalização graciosa do processo, isto foi feito tanto para Elixir quanto node. No K6 foi criado um script que executa um crash para 10% das requisições e posteriormente espera alguns segundos para fazer uma validação do *endpoint* de saúde, assim vendo se a aplicação reiniciou neste tempo e se ela continua a funcionar. Como o Node.js não se recupera automaticamente de falhas, foi utilizado a biblioteca *pm2* que faz o gerenciamento de serviços node e busca mantê-los operacionais.

Os resultados dos testes demonstrados no Quadro 5 mostram uma clara diferença na forma como Node.js e Elixir lidam com condições adversas. Node.js, não

conseguiu se recuperar a tempo, portanto houve diversas falhas nos testes e alguns distúrbios nos dados, como por exemplo a duração da requisição foi extremamente curta por conta de o K6 só conseguir realizar requisições completas em alguns momentos, ou também a inconsistência na quantidade de requisições que se dá muito provavelmente pelo número de tentativas feitas pelo script enquanto a aplicação estava fora do ar. Isso indica que, sob condições de falhas não tratadas ou inesperadas, Node.js pode não ser confiável para sistemas que necessitam de alta resiliência.

Por outro lado, Elixir demonstrou uma maior estabilidade conforme todos os seus testes foram bem-sucedidos, os distúrbios no total de requisições não foram observados e o tempo de resposta foi próximo do observado em outros experimentos, sendo um pouco mais oneroso do que o Node. Isso demonstra que uma das maiores forças do Erlang de fato é observada na prática, refletindo a robustez e a capacidade do Elixir de lidar com falhas de maneira mais eficaz, sem comprometer a estabilidade do sistema.

Considerando esses fatores, se a resiliência e a capacidade de recuperação de falhas são prioridades, Elixir parece ser a escolha mais adequada. No entanto, se o foco for na latência e no processamento rápido de requisições sob condições normais, Node.js pode ser mais vantajoso.

Quadro 5 – Resultados coletados de testes de tolerância a falhas usando o K6.

Linguagem	Métrica	Valor	Unidade
Node.js	Testes com sucesso	4,85	%
	p (95) Tempo de resposta	778,7	Microssegundos
	Falhas de requisição	95	%
	Requisições	67.026	Soma
Elixir	Testes com sucesso	100	%
	p (95) Tempo de resposta	17,41	Milissegundos
	Falhas de requisição	0	%
	Requisições	3.729	Soma

Fonte: Elaboração Própria a partir de k6 e código local.

7.4. Experimento de concorrência

Com o objetivo de testar concorrência foi utilizado o Redis rodando em memória na mesma máquina onde foi executado as aplicações Elixir e Node, foram utilizados três endpoints REST: um para criação dos itens no banco, um para update dos mesmos e por fim um para acessar os dados do banco baseado no identificador do item. O script do K6 por sua vez criava 100 itens no banco utilizando o *endpoint /api/item*, e posteriormente escolhia aleatoriamente uma das chaves para atualizar na rota */api/update/:key*, por fim realizava um *get* em */item/:key* para verificar se o item que está no banco de fato foi atualizado a tempo. Aqui também foi utilizado a

estratégia de rampas para que o número de requisições aumente conforme o tempo.

Como observado nos testes anteriores e ilustrado no Quadro 6, o Node.js obteve um tempo de resposta melhor que o Elixir, por conta de seu modelo I/O orientado a eventos, demonstrando ser muito eficiente em lidar com altas cargas de requisições. A maioria dos testes passaram, o que indica uma alta taxa de sucesso e confiabilidade.

O Elixir, por outro lado, apresentou um tempo de resposta maior, também observado em outros testes, atribuído ao seu modelo de gerenciamento de processos. Embora o tempo de resposta seja maior do que o de Node.js, a diferença não é excessivamente grande. Porém os testes falharam um pouco mais do que observado na aplicação node, o que ainda se encontra dentro de uma margem aceitável para muitos sistemas e que também possa ser melhorado quando obtido um conhecimento maior da tecnologia e de suas vantagens.

Quando se trata de concorrência a escolha entre Elixir e Node.js dependerá dos requisitos específicos do seu sistema. Se a latência baixa e a alta confiabilidade são críticas, talvez o Node.js possa ser a melhor escolha. No entanto, se a escalabilidade e a capacidade de gerenciar muitos processos simultâneos são mais importantes, Elixir pode oferecer benefícios significativos, especialmente com ajustes adequados na configuração de sua máquina virtual e outras otimizações específicas na implementação.

Quadro 6 – Resultados coletados de testes de tolerância a falhas usando o K6.

Linguagem	Métrica	Valor	Unidade
Node.js	Testes com sucesso	229.222	Soma
	Testes com falha	28	Soma
	p (95) Tempo de resposta	11,99	Milissegundos
	Requisições	91.720	Soma
Elixir	Testes com sucesso	222.554	Soma
	Testes com falha	271	Soma
	p (95) Tempo de resposta	30,55	Milissegundos
	Requisições	89.150	Soma

Fonte: Elaboração Própria a partir de k6 e código local.

8. Conclusão

Este estudo testou duas linguagens de *backend*, Elixir e Node.js, para o desenvolvimento de sistemas de IoT baseados em nuvem, focando na escalabilidade, desempenho assíncrono, tolerância a falhas e concorrência. Em quatro casos de teste, o programa de teste de carga K6 mediu as duas linguagens em vários cenários. Os resultados mostraram que cada linguagem tem prós e contras que impactariam a escolha da tecnologia em um sistema de IoT, dependendo do contexto.

O Node.js teve um desempenho melhor em termos de latência, apresentando latências muito menores em operações assíncronas e cenários de escalabilidade, como mostrado no texto. Isso se deve ao seu modelo de I/O não bloqueante e dirigido

por eventos. Além disso, a comunidade notavelmente maior e mais ativa, juntamente com seu rico ecossistema de bibliotecas, também torna o Node.js, uma escolha amplamente aceita e bem suportada para aplicativos web e implementações simples na nuvem.

Além disso, quando falamos sobre implementações em ambientes organizacionais, orientado a lucros, simplicidade, popularidade e facilidade de uso podem ser determinantes importantes na escolha da tecnologia. Portanto, o Node pode ser uma vantagem em implementações de baixa complexidade e quando a disponibilidade de recursos humanos é uma consideração.

O Elixir se destacou por sua resiliência e capacidade de lidar melhor com falhas. Finalmente, sob o modelo de falha tolerante, o Elixir demonstrou superar o Node quando se trata de tolerância a falhas, a aplicação permaneceu operacional sob carga extrema. Sua arquitetura baseada em processos leves usando gerenciamento eficiente de concorrência pode ser vista como uma vantagem potencial em sistemas distribuídos com necessidades de alta disponibilidade. No entanto, o Elixir apresentou tempos de resposta ligeiramente mais altos e falhas em alguns testes de concorrência, indicando a necessidade de ajustes e otimizações para alcançar um melhor desempenho.

Dessa forma, a escolha entre Elixir e Node.js sempre dependerá do ambiente e dos objetivos organizacionais, pois o primeiro pode ser mais adequado para cenários que requerem baixa latência, alta taxa de requisições, e que precisam ser implementados mais rapidamente, reduzindo o investimento em tempo de desenvolvimento, beneficiando-se de sua popularidade juntamente com o vasto ecossistema de bibliotecas. Ao mesmo tempo, o último pode oferecer mais vantagens em termos de escalabilidade e tolerância a falhas em relação ao Node.js, sendo uma opção robusta para sistemas que requerem alta disponibilidade ou a capacidade de gerenciar milhares de processos independentes. Porém, em um ambiente corporativo,

a comunidade menor em torno do Elixir e a curva de aprendizado mais acentuada são desvantagens, especialmente em relação à disponibilidade de mão de obra.

Assim, podemos concluir que ambas as linguagens geralmente têm suas áreas de especialização e podem satisfazer os requisitos de *backend* em nuvem para aplicativos de IoT. No entanto, a decisão final deve ser tomada dependendo das particularidades do projeto, como desempenho, resiliência, o tamanho da comunidade de código aberto, suporte e manutenção viáveis a longo prazo, bem como as peculiaridades do dispositivo específico a ser conectado à nuvem.

Referências

ANDERSEN, Jesper Louis. jlouis/fuse: A Circuit Breaker for Erlang, 2024. Disponível em: https://github.com/jlouis/fuse. Acesso em: 5 jun. 2024, às 17h38 min.

AUTH0. auth0/jsonwebtoken: About JsonWebToken implementation for node.js, 2024. Disponível em: https://github.com/auth0/node-jsonwebtoken. Acesso em: 5 jun. 2024, às 17h39 min.

BECKMANN, Julius. h4cc/awesome-elixir: A curated list of amazingly awesome Elixir and Erlang libraries, resources and shiny things, 2024. Disponível em: https://github.com/h4cc/awesome-elixir#other-awesome-lists. Acesso em: 5 jun. 2024, às 17h40 min.

BORBÉLY, Hunor Márton. How TypeScript Helps You Write Better Code, FreeCodeCamp, Nov 3. 2023 Disponível em: https://www.freecodecamp.org/news/benefits-of-typescript/. Acesso em: 3 jun. 2024, às 22h08 min.

BUYYA, Rajkumar; DASTJERDI, Amir Vahid. Internet of things: principles and paradigms. Amesterdã: Elsevier Science, 2016.

CMU SCHOOL OF COMPUTER SCIENCE. The "Only" Coke Machine on the Internet, The Carnegie Mellon University Computer Science Department, 1998 - 2024. Disponível em: https://www.cs.cmu.edu/~coke/history_long.txt. Acesso em: 3 jun. 2024, às 22h21 min.

CRYPTO. Node.js v22.2.0 documentation – Crypto, OpenJS Foundation, 2024. Disponível em: https://nodejs.org/api/crypto.html#crypto. Acesso em: 2 jun. 2024, às 22h33 min.

DOERRFELD, Bill. Key Insights From the Stack Overflow 2023 Developer Survey, Techstrong Group, 11 jul. 2023. Disponível em: https://devops.com/key-insights-from-the-stack-overflow-2023-developer-survey/. Acesso em: 2 jun. 2024, às 22h43 min.

ELIXIR WIKI. Elixir Libraries, MediaWiki, 2023-2024. Disponível em: https://www.elixirwiki.com/wiki/Elixir_Libraries. Acesso em: 3 jun. 2024, às 22h47 min.

ELIXIR WIKI. Erlang VM, 2023-2024. Disponível em: https://www.elixirwiki.com/wiki/Erlang_VM. Acesso em: 2 jun. 2024 às, 22h49 min.

ELIXIR-ECTO. elixir-ecto/ecto: A toolkit for data mapping and language integrated query, 2024. Disponível em: https://github.com/elixir-ecto/ecto. Acesso em: 5 jun. 2024, às 17h41 min.

ELIXIR-LANG. elixir-lang/elixir: Elixir is a dynamic, functional language for building scalable and maintainable applications, 2024. Disponível em: https://github.com/elixir-lang/elixir. Acesso em: 5 jun. 2024, às 17h46 min.

ERLANG. Erlang, 2024. The Erlang Programming Language. Disponível em: https://www.erlang.org/. Acesso em: 17 out. 2023, às 17h47 min.

EXPRESSJS. expressjs/express: Fast, unopinionated, minimalist web framework for node, 2024. Disponível em: https://github.com/expressjs/express. Acesso em: 5 jun. 2024, às 17h47 min.

FEDRECHESKI, Geovane; COSTA, Laisa Caroline de Paula; ZUFFO, Marcelo Knörich. Elixir programming language evaluation for IoT, IEEE International Symposium on Consumer Electronics (ISCE), 2016.

GITHUB. Let's build from here - The world's leading AI-powered developer platform, GitHub, Inc, 2008-2024. Disponivel em: https://github.com/. Acesso em: 4 jun. 2024, às 12h06 min.

GRAFANA LABS. Grafana/k6: A modern load testing tool, using Go and JavaScript, 2024. Disponível em: https://github.com/grafana/k6. Acesso em: 5 jun. 2024, às 17h57 min.

HAENISCH, Till. A case study on using functional programming for internet of things applications, Athens Journal of Technology & Engineering, 2016. Disponível em: https://www.athensjournals.gr/technology/2016-3-1-2-Haenisch.pdf. Acesso em: 4 jun. 2024, às 23h10 min.

HAPI.JS. hapijs/joi: The most powerful data validation library for JS, 2024. Disponível em: https://joi.dev/. Acesso em: 5 jun. 2024, às 17h35 min.

HEX. The package manager for the Erlang ecosystem, Six Colors AB, 2020-2024. Disponível em: https://hex.pm/. Acesso em: 4 jun. 2024 às 11h59 min.

HONEYPOT. Elixir: The Documentary, 13 jul. 2018, In.: Youtube, 2018-2024. Disponível em: https://www.youtube.com/watch?v=lxYFOM3UJzo. Acesso em: 4 jun. 2024, às 17h22 min.

IBM. What is the internet of things?, IBM, 2 de Jun. de 2024. Disponível em: https://www.ibm.com/topics/internet-of-things. Acesso em: 5 Jun 2024, às 17h51 min.

JAISINGHANI, Dheryta; SINGH, Gursimran; FULARA, Harish; MAITY, Mukulika; NAIK, Vinayak. Elixir: Efficient data transfer in WiFi-based IoT nodes. Proceedings of the 24th Annual International Conference on Mobile Computing and Networking. Anais. New York, NY, USA: ACM, 2018.

KASHCHA, Andrei. NPM Rank, Github Gists, 2024. Disponível em: https://gist.github.com/anvaka/8e8fa57c7ee1350e3491. Acesso em: 2 jun. 2024 às 23h00 min.

KELEKTIV. kelektiv/node.bcrypt.js: bcrypt for NodeJs, 2024. Disponível em: https://github.com/kelektiv/node.bcrypt.js. Acesso em: 5 jun. 2024 às 17h48 min.

KIDD, Chrissy. What Is the CIA Security Triad? Confidentiality, Integrity, Availability Explained, BMC Software, 24 nov. 2020,. Disponível em: https://www.bmc.com/blogs/cia-security-triad/. Acesso em: 3 jun. 2024 às 23h03 min.

MARTIN, Robert Cecil. Clean code: A handbook of agile software craftsmanship. Filadélfia, PA, USA: Pearson Education, 2009.

MATAB, Saif. Exploring Event-Driven and Asynchronous Programming in Node.js, Dev.to, 12 mai. 2024, Disponível em: https://dev.to/kernelrb/exploring-event-driven-and-asynchronous-programming-in-nodejs-3clc. Acesso em: 3 jun. 2024 às 23h18 min.

NISHIGUCHI, Masatoshi. IoT development using Raspberry Pi, Elixir and Nerves, 2 de abr. de 2021. Disponível em: https://dev.to/mnishiguchi/iot-development-using-rapberry-pi-and-elixir-iij. Acesso em: 4 jun. 2024 às 14h45 min.

NODE. nodejs/node: Node.js JavaScript runtime, 2024. Disponível em: https://github.com/nodejs/node. Acesso em: 5 jun. 2024 às 17h45 min.

NPM. Build amazing things, Npm Inc, 2014-2024. Disponível em: https://www.npmjs.com/. Acesso em: 4 jun. 2024 às 12h04 min.

OPENJS FOUNDATION. An introduction to the NPM package manager, 2012-2024. Disponível em: https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager. Acesso em: 3 Jun 2024 às 23h06 min.

OPENJS FOUNDATION. JavaScript Asynchronous Programming and Callbacks, 2012-2024. Disponível em: https://nodejs.org/en/learn/asynchronous-work/javascript-asynchronous-programming-and-callbacks. Acesso em: 3 Jun 2024 às 23h06 min.

OPENJS FOUNDATION. Node.js v20.2.0 Documentation, 2011-2024. Disponível em: https://nodejs.org/docs/latest/api/. Acesso em: 3 jun. 2024 às 22h41 min.

OPENJS FOUNDATION. The Node.js Event Loop, 2012-2024. Disponível em: https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick. Acesso em: 3 Jun 2024 às 23h07 min.

OPENJS FOUNDATION. The V8 JavaScript Engine, 2012-2024. Disponível em: https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine. Acesso em: 3 Jun 2024 às 23h08 min.

OWASP. Internet of Things (IoT) Top 10 - 2018 Final, The OWASP IoT Security Team, 2018. Disponível em: https://wiki.owasp.org/images/1/1c/OWASP-IoT-Top-10-2018-final.pdf. Acesso em: 3 jun. 2024 às 23h20 min.

PADMANABHAN, Arvind; Arjun. Programming for IoT, Devopedia, 12 de aug. de 2020. Disponível em: https://devopedia.org/programming-for-iot. Acesso em: 16 out. 2023 às 22h35 min.

PFLEEGER, Charles.; PFLEEGER, Shari. L.; COLES-KEMP, Lizzie. Security in computing. 6. ed. Boston, MA, USA: Addison Wesley, 2024.

PHOENIX FRAMEWORK. phoenixframework/phoenix: Peace of mind from prototype to production, 2024. Disponível em: https://github.com/phoenixframework/phoenix. Acesso em: 5 jun. 2024 às 17h49 min.

REMY. remy/nodemon: Monitor for any changes in your node.js application and automatically restart the server - perfect for development, 2024. Disponível em: https://github.com/remy/nodemon. Acesso em: 5 jun. 2024 às 17h50 min.

SCHOENFELDER, Paul. bitwalker/libcluster: Automatic cluster formation/healing for Elixir applications, 2024. Disponível em: https://github.com/bitwalker/libcluster. Acesso em: 5 jun. 2024, às 17h50 min.

STRZELEWICZ, Alexandre. Unitech/pm2: Node.js Production Process Manager with a built-in Load Balancer., 2024. Disponível em: https://github.com/Unitech/pm2. Acesso em: 5 jun. 2024, às 17h54 min.

THE ELIXIR TEAM. Kernel, Elixir v1.16.3 Documentation, 2012-2024. Disponível em: https://hexdocs.pm/elixir/1.16.3/Kernel.html. Acesso em: 3 jun. 2024 às 22h38 min.

THE LIVING INTERNET. ARPANET - The First Internet, 2000-2024. Disponível em: https://www.livinginternet.com/i/ii_arpanet.htm. Acesso em: 3 jun. 2024, às 22h05 min.

UEBERAUTH. ueberauth/guardian: Elixir Authentication, 2024. Disponível em: https://github.com/ueberauth/guardian. Acesso em: 3 jun. 2024, às 21h55 min.

VALIM, José. Introduction to Elixir: a New Language on the VM - José Valim, 18 de mar. de 2013. In.: Youtube, 2010-2024. Disponível em: https://www.youtube.com/watch?v=41PvAPSX0wg. Acesso em: 3 jun. 2024, às 21h54 min.

WHITLOCK, David. riverrun/comeonin: Password hashing specification for the Elixir programming language, 2024. Disponível em: https://github.com/riverrun/comeonin. Acesso em: 2 jun. 2024, às 17h55 min.

ZENNER IOT SOLUTIONS. ZennerIoT/ex_audit: About Ecto auditing library that

transparently tracks changes and can revert them, 2024. Disponível em: https://github.com/ZennerIoT/ex_audit. Acesso em: 2 jun. 2024, às 17h56 min.