ChatGPT

Analogias “Quentes” em Espaços Conceituais Multi-camadas

A) Mapa de Dúvidas e Lacunas (GAP MAP)

Para garantir um projeto robusto, listamos as principais **lacunas e questões** agrupadas por área, com impacto e urgência estimados, possíveis soluções e fontes sugeridas para investigação.

Teoria e Definições

Lacuna/Questão Teórica	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Definição de “analogia quente” – O conceito ainda vago: o que exatamente caracteriza uma analogia “sem quinas” e de alto *hotness*?	Alto	Agora	Revisar literatura de analogias (Gentner, Holyoak) e formalizar atributos essenciais (p.ex. mapeamento relacional completo, sem contradições).	Gentner (1983) [1]; Holyoak & Thagard (1995)
Coerência analógica – Critérios objetivos para “sem quinas” (sem arestas desconexas ou conflitos) ainda não claros. SMT sugere *systematicity*, mas há críticas.	Alto	Agora	Analisar *Systematicity Principle* [2] e seus limites. Definir métrica de coerência (ex.: % de relações mapeadas sem exceção).	Gentner (SMT) [1]; Lee & Holyoak (2008) [3]
“Hotness” como métrica – Como decompor o *hotness* em dimensões mensuráveis (semântica, estrutural, funcional, escala)? Falta uma base teórica unificada.	Alto	Agora	Desenvolver vetor de dimensões com base em literatura de similaridade (vetores semânticos, isomorfismo de grafos, papel cibernético, etc.). Validar se a combinação faz sentido (experimentos).	Turney (2008) [4] [5]; Ashby (1956) [6]

Lacuna/Questão Teórica	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Papel cibernético (VSM) nas analogias – Como definir rigorosamente "papel funcional equivalente" entre domínios (ex.: *System 3* em software vs. biologia)?	Médio	Depois	Mapear funções VSM S1–S5 de casos conhecidos (Cybersyn, Herring 2007). Formalizar tags de papel (ex.: sensor, atuador) e verificar consistência no grafo.	Beer (1972); Gorelkin (2025) [7] [8]
Escala temporal e variedade – Conceitos de timescale e variedade de Ashby não estão bem integrados no modelo de analogia.	Médio	Depois	Incluir meta-dados de escala (tempo de ciclo, frequência) e variedade (nº estados controlados) em nós. Examinar se analogias exigem matching de escala (ex.: ambos tratam eventos por minuto).	Ashby (1956) [6]; Heylighen (2003) [9]

Modelagem de Dados (Grafos/Ontologias/Camadas)

Lacuna/Questão de Modelagem	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Formato canônico de grafo – Precisamos especificar um GraphSpec unificado. Quais tipos de arestas e metadados incluir?	Alto	Agora	Definir um esquema JSON para nós/arestas com campos necessários (id, rótulo, domínio, camada, tags, evidências). Garantir extensibilidade para novas relações.	Padrões de ontologia (OWL/RDF); JSON-LD spec
Extração de subconceitos – Como extrair subestruturas de diferentes fontes (texto, código, ontologia) de forma consistente?	Alto	Agora	Implementar **importadores** específicos (NLP para texto, AST parser para código, parser RDF) que produzam GraphSpec compatíveis. Normalizar nomes (sinônimos para mesmo conceito).	Ontology Matching [10]; Ferramentas AST Python

Lacuna/Questão de Modelagem	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Multi-camadas vs. explosão combinatória – Há risco de grafo gigantesco (cada conceito expandindo recursivamente). Como limitar camadas sem perder essência?	Alto	Agora	Aplicar limites de profundidade **d** e filtragem por relevância (top-K subconceitos por importância). Usar heurísticas: exibir até N filhos mais centrais, consolidar similares.	Network pruning métodos; OAEI Anatomy (limitar classes)
Camada "Org/Cyber" comum – A camada de metaconceitos (VSM, feedback loops) é opcional. Como incorporá-la sem forçar analogias irrelevantes?	Médio	Depois	Tornar "Org/Cyber" um conjunto de tags (não nós explícitos), anexando em nós de outros domínios quando aplicável. Permitir analogias diretas ou via comparação das tags de papel.	VSM em software [11] [12] ; Patterns em DevOps
Normalização de rótulos e IDs – Mesmos conceitos podem aparecer de formas distintas nas fontes. Risco de nós duplicados ou não reconhecidos.	Médio	Agora	Implementar função de normalização (ex.: lower case, snake_case). Usar identificadores únicos estáveis (URI para ontologias, hash para texto). Merge de nós duplicados via heurística semântica.	String similarity (Levenshtein); Ontology ID best practices

Algoritmos (Matching/Alinhamento/Scoring)

Lacuna/Questão de Algoritmos	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Matching guloso vs. ótimo – MVP usará casamento guloso, mas isso pode perder alinhamentos globais ótimos. Quando introduzir algoritmos de otimização (Hungarian, ILP)?	Alto	Agora (MVP simples), v2 para otimização	Iniciar com heurísticas rápidas (grafo menor). Para v2, integrar algoritmo de atribuição ótima em subgrafos candidatos. Testar diferença de qualidade para decidir migração.	Algoritmo Húngaro (Kuhn-Munkres); ILP matching papers

Lacuna/Questão de Algoritmos	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Penalidades por inconsistência relacional – Como formalizar punição quando conceitos se alinham mas relações adjacentes não batem?	Alto	v2	Introduzir função de custo que subtrai pontos se uma correspondência sugerir relações incompatíveis (ex.: A causa B vs. X análogo causa Y? Se não, penalizar). Calibrar com ground truth.	Ontology alignment consistency [10]; Constraint solving (ILP)
Escalabilidade do matching – Comparar todos pares de nós entre dois grafos camadas profundas é O(N^2). Risco de lentidão.	Alto	Agora	Aplicar filtragem inicial: só considerar pares com similaridade semântica acima de threshold ou compartilhando tags. Cachear embeddings e usar *nearest neighbors* para limitar candidatos.	Node2Vec embeddings; Annoy/FAISS (busca vetorial)
Aprendizado de máquina (v3) – Vale usar GNNs ou aprendizado supervisionado para melhorar o score? Carece de ground truth robusto.	Médio	Depois	Explorar GNN que encodifica subgrafos e verifica correspondência estrutural. Necessário dataset de analogias verdadeiras para treinar – possivelmente gerado a partir de ontologias alinhadas ou casos confirmados.	GNN alignment [13]; BERTMap (aprende correspondências) [14] [15]
Matching incremental/local – Com novas informações surgindo (ex.: código atualizado), recomputar tudo é caro.	Médio	Depois	Implementar *pipeline* incremental: recalcular score só em partes afetadas (ex.: se apenas um nó mudou, refazer analogias envolvendo ele). Manter cache de correspondências estáveis.	Incremental graph algorithms; diff de ontologias

Validação (Ground truth, Métricas, Estudos)

Lacuna/Questão de Validação	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Falta de *ground truth* – Não há dataset "oficial" de analogias bio↔software rotuladas. Como validar se o heatmap está correto?	Alto	Agora	Coletar casos de analogias reconhecidas (ex.: apoptose ~ circuit breaker) e pedir a especialistas pontuarem. Usar ontologias alinhadas conhecidas (FMA-SNOMED) para avaliar componente ontológico.	OAEI benchmarks [10] ; Estudos de caso na literatura de bio/software [16] [17]
Métricas de sucesso – Precisamos definir precision/recall de *mappings*, mas também métricas de utilidade (análogo gerou insight útil?).	Alto	Agora	Medir *precision/recall* comparando com alinhamentos esperados (onde houver). Criar métrica de *usefulness* qualitativa: após ver analogia, usuário tomou decisão correta? Avaliar em estudos controlados.	Ontology alignment eval (Precision/Recall) [18] ; UX research (utilidade percebida)
Consistência por camada – Uma analogia pode ser forte em conceitos gerais (camada 0) mas falhar nos detalhes (camada 2+). Como refletir isso?	Médio	Agora	Calcular *hotness* por camada (Hot_0, Hot_1, ...). Exigir que analogia "quente" tenha escores altos nas primeiras camadas e não caia abruptamente. Sinalizar onde há quebras ("divergências na camada d").	Gentner (mapeamento profundo) [1] ; Análise hierárquica de analogias
Estabilidade temporal – Se o sistema evolui (código refatorado ou novo dado), o score deve mudar gradualmente, não aleatoriamente.	Médio	Depois	Testar em versões de um mesmo sistema (ex.: antes e depois refatoração): analogias fundamentais devem persistir ou mudar de forma explicável. Introduzir inércia nos scores (ex.: média móvel).	Análise de versão de software [19] [20] ; Concept drift detection

Lacuna/Questão de Validação	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Detecção de analogias perigosas – Como identificar e marcar analogias que *parecem* quentes mas na verdade enganam (falsos positivos)?	Alto	Agora	Definir critérios de alerta: se analogia tem alta estrutura mas conteúdo causal divergente (ex.: correlação vs causalidade diferente) [21], sinalizar "risco". Incluir validação humana obrigatória para scores muito altos em domínios distantes.	Críticas à systematicity cega [2] [22]; Goodhart aplicado a analogias

Visualização e UX

Lacuna/Questão de Visualização/UX	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Render de matrizes grandes – Um heatmap de, digamos, 1000x1000 (um grafo robusto vs outro) pode ser lento e ilegível.	Alto	Agora	Usar WebGL ou WebGPU para rasterizar a matriz em GPU, garantindo ~60fps [23] [24]. Implementar *level-of-detail*: agrupar células em clusters em zoom out, detalhar ao dar zoom.	*Shinyheatmap* (100k+ linhas) [23] [24]; WebGL heatmap (Mapbox)
Tooltip e destaque cruzado – Precisamos de explicações ao passar mouse em uma célula (analogia específica) e destaque de nós relacionados.	Alto	Agora	Implementar tooltip fixo mostrando: conceitos correspondentes + score local + top relações alinhadas/ diferentes. Ao focar, destacar esses nós e arestas no grafo original (ex.: borda brilhante nos nós envolvidos).	Princípio de *brushing and linking* em vis [25]; D3.js linking examples
Cores e percepção – Escolha do colormap é crítica: escalas *rainbow/jet* distorcem dados e prejudicam daltônicos [26].	Alto	Agora	Adotar paleta perceptualmente uniforme (ex.: Viridis ou inferno). Testar contraste para daltônicos [27]. Evitar usar vermelho-verde juntos. Fornecer opção de view em escala de cinza para impressão.	Nature (2020) sobre cores [26] [28]; Guidelines W3C acessibilidade

Lacuna/Questão de Visualização/UX	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
SVG vs Canvas vs WebGL – O que usar para desenhos de subgrafos e UI? Cada tecnologia tem trade-offs de performance e nitidez.	Médio	Agora	Heatmap principal: Canvas/ WebGL (muitas células). Elementos UI e gráficos pequenos: SVG (mais nítido para texto e escalável). Combinar ambos conforme necessidade (*hybrid rendering*).	Experiências em vis web grandes (HiGlass, deck.gl docs)
Interação multi-camada – Como permitir que usuário navegue por camadas (0,1,2...)?	Médio	Depois	Fornecer um slider ou controle de profundidade. Ao ajustar, recalcular ou filtrar heatmap mostrando score apenas até aquela profundidade (Hot_d). UI deve deixar claro quando info é *partial*.	Design de dashboards hierárquicos; Approach LOD (level of detail)

Riscos e 2ª Ordem

Lacuna/Questão de Risco	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
Efeito Goodhart – Se times utilizarem o score de "hotness" como meta, podem aparecer distorções (forçar analogias ao invés de melhorar sistema).	Alto	Agora	*Não* apresentar hotness isolado como KPI de sucesso organizacional. Usar apenas como ferramenta exploratória. Incluir alertas sobre contexto e limites do score. Monitorar comportamentos de "gaming" (ex.: code changes só para aumentar score).	Goodhart's Law (1975) – *"quando uma medida vira meta, deixa de ser boa medida"* [29]; Case studies métricas em org.
Enviesamento de dados – Ontologias ou texto podem conter vieses (ex.: analogias baseadas em conceitos populares mas não relevantes).	Médio	Depois	Auditar conjunto de conceitos iniciais para diversidade. Permitir feedback do usuário para ajustar pesos se analogia parece irrelevante. Incorporar fontes diversas (biologia, software de vários domínios) para balancear.	Debiasing em AI; Diversidade taxonômica (ontologias múltiplas)

Lacuna/Questão de Risco	Impacto	Urgência	Como Resolver (ação)	Fontes Sugeridas
"Metáfora enganosa" – Usuários podem achar uma analogia charmosa e aplicar indevidamente (ex.: *apoptose* em contexto errado).	Alto	Agora	Fornecer, junto com cada analogia, uma seção "Onde a analogia quebra" listando diferenças não mapeadas. Incentivar validação empírica de qualquer insight antes de adoção (experimentos ou protótipos).	Literatura de analogias em ciência (Hesse, 1966 – uso e abuso de analogias); Explicabilidade XAI
Sobrecarga cognitiva – Visual de heatmap grande + grafo + painel de explicação pode sobrecarregar usuário.	Médio	Agora	Aplicar design minimalista: mostrar o necessário por padrão (ex.: só top 10 analogias relevantes), com opção de explorar detalhes sob demanda. Usar indicadores visuais simples (células quentes chamativas, paleta discreta para frias).	Princípios de UX (Nielsen); Tufte – evitar *chartjunk*.
Segurança e privacidade (local-first) – Indexar repositórios e textos locais implica em manusear dados sensíveis (código proprietário).	Médio	Agora	Garantir que toda a análise fica offline (sem chamadas externas). Possibilitar criptografar o armazenamento local (SQLite) ou permitir opt-out de certos arquivos. Documentar claramente que nenhum dado sai da máquina.	Normas de segurança de software; OWASP para apps desktop

B) Fundamentos: Definições "Sem Quinas"

Para construir um modelo sólido, precisamos de definições claras dos principais conceitos teóricos:

- **Analogia (estrutural/relacional, não apenas semântica):** Em vez de tratar analogia como simples similaridade superficial entre dois domínios, adotamos a visão de **similaridade relacional estruturada** conforme a *Structure Mapping Theory (SMT)* de Dedre Gentner [1]. Uma analogia significa mapear uma estrutura de relações entre um **domínio-fonte** e um **domínio-alvo**, ignorando diferenças de atributos irrelevantes [30] [31]. Por exemplo, dizer que *"o átomo é como o sistema solar"* foca nas relações (eletros giram em torno do núcleo, planetas em torno do sol) e não nas diferenças de tamanho ou composição [32]. **Boa analogia** implica correspondência consistente entre múltiplas relações – e não apenas uma semelhança isolada – seguindo o princípio da *sistematicidade*, que favorece *conjuntos interconectados de relações* [33] [34]. Em resumo, analogia aqui é vista como um **mapeamento parcial de grafo**, alinhando nós e arestas de modo a preservar a estrutura relacional mais profunda possível.

- **Coerência analógica ("sem quinas"):** Chamamos de analogia **sem quinas** aquela cujo mapeamento é suave e completo, sem arestas "sobrando" ou contradições gritantes. Operacionalmente, isso significa que o conjunto de correspondências cobre os principais elementos de cada domínio *sem gerar conflitos lógicos*. Em termos de grafo, todas as relações-chave de um lado encontram imagem no outro lado (mesmo tipo de relação ou análoga) – não há uma relação no domínio-fonte que corresponda a nada no alvo (*gap*), nem relações mapeadas de forma cruzada (incompatível). A *SMT* propõe que a coerência advém de mapear preferencialmente estruturas *profundas e interligadas* (princípio da sistematicidade) [35]. Entretanto, coerência não é binária; há graus. Podemos quantificar coerência pelo percentual de relações preservadas no mapeamento e ausência de conflito. Por exemplo, se 90% das relações de A (domínio-fonte) encontram correspondentes em B (domínio-alvo) e vice-versa, e nenhuma implicação importante de A contradiz B, temos alta coerência. Vale notar que nem sempre analogias de alta coerência são válidas ou úteis – mas se forem incoerentes, certamente serão analogias ruins. **Critério "sem quinas"**: todos os elementos mapeados se encaixam, sem precisar "forçar" a correspondência de algo que claramente não combina (quina seria uma aresta que não pode ser suavizada). Na prática, definiremos um **score de coerência** que penaliza severamente relações não mapeadas ou mal alinhadas.

- **Hotness (calor da analogia):** Conceituamos *hotness* como um **vetor multidimensional de escores** que captura diferentes aspectos da qualidade da analogia, além de um **score composto** resultante. As dimensões principais incluem: (a) **Semântico** – similaridade de significados entre conceitos alinhados, medido via embeddings ou comparação lexical; (b) **Estrutural/Relacional** – quanto da estrutura de relações de um domínio é preservada no outro (ex.: padrão de rede de interações, causalidade, hierarquia); (c) **Funcional (papel cibernético)** – alinhamento de funções sistêmicas (ex.: elemento no sistema A que atua como sensor corresponde a elemento sensor em B? Existe análogo de "controlador" em ambos?); (d) **Escala/Temporal/Falhas** – equivalência de escala de tempo e tipo de dinâmica (ex.: ambos operam em escala diária? ambos lidam com falhas repentinas vs gradativas?); e potencialmente (e) **Generatividade** – se a analogia sugere novas hipóteses/testes (uma medida mais subjetiva de valor heurístico). Cada dimensão vai de 0 a 1 (ou -1 a 1 se houver oposição) e podemos representar *hotness* como vetor $H = (H_{\text{sem}}, H_{\text{str}}, H_{\text{func}}, H_{\text{scale}}, H_{\text{gen}})$. O score composto pode ser uma média ponderada: $H_{\text{comp}} = \sum w_i H_i$, com pesos ajustáveis (inicialmente iguais, ou enfatizando dimensões conforme contexto). Uma analogia "quente" teria todos (ou a maioria) dos componentes com valor alto. Importante: *hotness* não é metáfora vaga, mas **medida operacional**. Por exemplo, analogias científicas bem-sucedidas geralmente têm alta similaridade estrutural e funcional, mesmo que semântica superficial baixa (bateria vs reservatório de água – compartilham relações de fluxo e armazenamento [36]). Mediremos cada componente de forma objetiva (ver seção E), o que permite rastrear por que uma analogia tem dado *hotness* – e oferecer explicações por dimensão (ex.: "alto calor estrutural, mas baixo em escala temporal").

- **Espaço conceitual multi-layer (multi-camadas):** Trata-se da representação canônica dos conhecimentos em formato de grafo hierárquico. Cada **conceito raiz** pode ter **subconceitos** (filhos) formando camadas de detalhe. Por exemplo, no domínio biologia um conceito *"Sistema Imune"* possui subconceitos como *"Imunidade Inata"* e *"Imunidade Adaptativa"*, e esses por sua vez possuem componentes (células NK, linfócitos B, etc.). As **relações** conectam nós, podendo ser *is-a* (hierárquica), *parte-de*, *causa*, *regula*, *comunica*, etc., incluindo relações cibernéticas (sensor, atuador, feedback). O grafo é **multi-camadas** pois podemos distinguir nível de abstração: camada 0 (conceitos gerais), camada 1 (subpartes principais), camada 2 (detalhes internos) e assim por diante.

Em cada camada, emergem "subárvores" conceituais. Uma analogia pode ser analisada camada a camada: às vezes dois sistemas se correspondem bem nos papéis gerais (camada 0), mas divergências aparecem em camadas profundas (detalhes). Vamos explicitar essas camadas para evitar confundir similaridade superficial com estrutural: a ferramenta permitirá filtrar/navegar por nível. **Importante:** os nós podem ser etiquetados com **metadados** indicando camada e até *domínio*. No projeto, prevemos três domínios principais (conforme placeholders preenchidos): **Biologia**, **Software** e **Organizações/Cibernética** (este último servindo como *ponte* conceitual – os papéis abstratos). Isso significa que no grafo multi-layer, alguns nós pertencem a Biologia, outros a Software, mas todos podem ser anotados também com tags de papel cibernético (ex.: um nó "célula apoptótica" tem tag *System 1 (operação local)* e *falha segura*, enquanto um nó "circuit breaker" em software tem tags análogas). Esse arranjo multi-layer e multi-domínio visa alinhar **macro conceitos** (camada 0, ex.: *morte celular programada* vs *mecanismo de desligamento gracioso*), **meso conceitos** (camada 1-2, ex.: *vias de sinalização da apoptose* vs *algoritmo de monitoramento do circuit breaker*) e **micro detalhes** (camada 3+, se disponível).

- **Papel/role cibernético:** Inspirados pelo *Viable System Model (VSM)* e pela teoria de controle, definimos um conjunto de **papéis funcionais** que elementos de sistemas podem exercer, independente do domínio. Exemplos: **Sensor** (monitora estado), **Atuador** (executa ação no sistema), **Controlador** (toma decisão/cálculo para corrigir desvio), **Referência/Setpoint** (objetivo ou padrão desejado), **Ruído/Perturbação** (fator externo), **Ciclo de feedback** (laço completo). No VSM de Stafford Beer, os sistemas são divididos em subsistemas S1 a S5 [37] [38]: *S1 Operações*, *S2 Coordenação*, *S3 Controle interno*, *S3* (3 estrela, auditória), *S4 Inteligência (futuro)*, *S5 Política*. Traduzindo para software/organizações: um microserviço pode ser visto como *S1* (faz trabalho), o *System 3* equivalente seria gestão de recursos (ex.: *orquestrador Kubernetes*), *System 4* seria uma função de planejamento/adaptação (ex.: auto-scaling preditivo olhando métricas históricas), e *System 5* representa governança/política (ex.: um arquiteto definindo políticas de toda a plataforma). Usaremos essas ideias como **tags** nos nós/entidades para ajudar no alinhamento funcional. Por exemplo, se no domínio biologia um elemento tem tag "System 2" (coordenação anti-oscilações, como talvez *células reguladoras T* evitando resposta imunológica exagerada) e em software temos um componente *rate limiter* (que suaviza picos de carga), essa tag comum indica possível analogia de *papel* – mesmo que semanticamente sejam distintos. Assim, *papel cibernético* é uma **abstração de função**: permite comparar sistemas por *o que cada parte faz*, não pelo nome ou estrutura em si. Será especialmente útil para encontrar analogias criativas (ex.: *apoptose* comparada a *circuit breaker*, ambos exercendo função de *auto-sacrifício para salvar o sistema*, um padrão de viabilidade).

- **Apoptose vs Necrose (como padrão de viabilidade):** Esses termos vêm da biologia e ilustram conceitos de *falha segura vs falha catastrófica*. **Apoptose** é a *morte celular programada*, um processo ordenado e benéfico onde a célula se autodestrói de forma controlada, minimizando dano aos vizinhos [39]. Já **Necrose** é morte celular *acidental/traumática*, descontrolada, que libera conteúdo tóxico causando inflamação e danos ao tecido ao redor [40]. Na perspectiva de sistemas viáveis, apoptose é um **mecanismo de robustez**: remove componentes defeituosos ou desnecessários do sistema sem comprometer o todo – análogo a desligar um microserviço de forma graciosa quando detectado problema (circuit breaker disparando, por exemplo). Necrose seria o oposto: um componente falha de forma **não isolada**, corrompendo dados ou travando recursos de forma que afeta todo o sistema (como um processo que trava e congestiona todo servidor porque não foi isolado). Portanto, apoptosis vs necrosis exemplifica um **padrão vs antipadrão** de viabilidade. Vamos usar essa analogia como **benchmark**: um caso concreto onde Biologia → Software mapeia

bem (apoptose ≈ desligamento gracioso, necrose ≈ pane catastrófica sem isolamento). Notar que apoptose envolve sinais internos e externos, cascatas bem definidas (via intrínseca/extrínseca envolvendo caspases) e produção de *apoptotic bodies* que são limpos pelo sistema imune [39] – enquanto necrose é "explosiva". Traduzindo: em software, um componente que implementa *graceful shutdown* se desliga em etapas, libera recursos, notifica outros para tomarem seu lugar (análogo às vesículas apoptóticas fagocitadas) em vez de simplesmente travar e corromper contexto (necrose). Definir claramente esses conceitos com base na literatura [40] nos ajuda a treinar nosso modelo e também comunicar aos usuários o *porquê* de certo padrão ser saudável ou não.

- **Requisite Variety (Lei da Variedade Requisita):** Formulada por W. Ross Ashby (1956), estabelece que *"só variedade absorve variedade"* [6] . Ou seja, para um sistema (ou controlador) lidar com as inúmeras condições do ambiente (*distúrbios*), ele próprio precisa ter um repertório de estados ou respostas de igual riqueza. Em termos práticos: se o ambiente pode apresentar 100 tipos de problemas, o sistema de controle precisa ter soluções para ~100 situações, caso contrário algo escapará. Vamos incorporar esse conceito como parte da análise funcional: dois sistemas em analogia devem ter "variedade compatível"? Por exemplo, um organismo enfrenta variedade de ameaças (vírus, bactérias, etc.) e tem sistema imune adaptativo (capaz de aprender variedade praticamente ilimitada de antígenos) – analogia em software: um sistema de segurança cibernética com capacidade adaptativa (machine learning detectando novos ataques) vs um baseado apenas em assinatura fixa. Um match analógico bom implicaria que ambos satisfazem (ou não) Ashby. No nosso modelo, *variedade* pode ser medida pelo número de estados distintos ou transições que o sistema pode acomodar [9] . Ao comparar dois sistemas, se um é muito mais simples (variedade baixa) que o outro, a analogia entre eles talvez seja rasa (ex.: analogia entre um termostato simples e um ecossistema pode falhar – escalas de variedade muito distintas). Portanto, incorporaremos *requisite variety* como atributo dos nós de nível sistêmico (camada 0): uma aproximação quantitativa ou categórica (baixa, média, alta variedade). Isso entra na dimensão *Timescale/Failure-mode* do *hotness*. Um par analógico ideal terá níveis compatíveis – ou se não, isso será apontado na explicação ("O sistema A cobre mais casos do que B – analogia pode falhar em algumas variedades não correspondentes").

- **Controle (setpoint, sensor, controlador, atuador, planta, etc.):** Base do pensamento cibernético de 1ª ordem, são conceitos padrão da teoria de controle feedback [41] [42] . Definições operacionais:

  - *Setpoint* (referência): o valor desejado da saída do sistema (ex.: temperatura ideal do forno).
  - *Sensor*: mecanismo que mede a saída real (ex.: termômetro lendo 180°C).
  - *Controlador*: calcula erro (diferença entre medido e desejado) e decide ação de correção (ex.: controlador PID aumentando gás se muito frio).
  - *Atuador*: componente que executa a ação determinada (ex.: válvula de gás abrindo/fechando).
  - *Planta/Processo*: o sistema sendo controlado (ex.: o forno em si, cuja temperatura muda conforme entrada de gás).
  - *Distúrbio*: influências externas indesejadas (ex.: ambiente frio roubando calor).
  - *Feedback*: laço formado sensor→controlador→atuador, ajustando continuamente o sistema [43] .

Esses elementos formam o **loop de controle clássico** [44] . No nosso grafo, muitos sistemas tanto biológicos quanto de software podem ser modelados assim. Por exemplo, o corpo humano regula temperatura: setpoint no hipotálamo, sensores termoceptores, atuadores musculares (tremor) ou sudorese, planta é o corpo. Em software, controle de *autoscaling*: setpoint de CPU 60%, sensor monitora uso, controlador decide

lançar mais instâncias, atuador (API cloud) inicia VM. Explicitar essa analogia permite alinhar diretamente função a função – se conseguirmos identificar os papéis equivalentes, a analogia fica muito **concreta**. Portanto, definiremos no *GraphSpec* relações tipo *senses/monitors*, *acts_on*, *setpoint_for*, *controlled_by* para capturar essas estruturas. E as tags de papel cibernético citadas acima (S1-S5) complementam: muitas vezes um loop de controle inteiro pode ser visto como subsistema S3 dentro de VSM (gestão interna), enquanto S4 lida com setpoints adaptativos. Em suma, entender e formalizar controle nos ajuda a reconhecer padrões análogos (ex.: *circuit breaker* não é exatamente um controlador, mas age como atuador autônomo desligando circuito quando sensor (contador de falhas) ultrapassa threshold – *um laço de controle local com setpoint fixo*). Vamos, portanto, **comparar sistemas também por sua arquitetura de controle**, alinhando loops a loops quando possível.

> **Observação:** Muitas dessas definições têm visões concorrentes na literatura. Por exemplo, teorias de analogia incluem Gentner (estrutural) vs. Hofstadter (ênfase em criatividade – *Copycat*) e modelos conexionistas (LISA de Hummel & Holyoak). No escopo aqui, escolhemos a abordagem SMT (Gentner) por fornecer critério formal claro (relacional) [1]. Onde há divergências (como a crítica de que *sistematicidade* nem sempre garante analogia útil [2] [21]), adotamos postura pragmática: **se a literatura não fornece consenso, implementamos uma métrica parametrizável** (ex.: peso de sistematicidade) que pode ser ajustada ou testada empiricamente com usuários. A filosofia é combinar rigor científico *e* aplicabilidade: se definições precisas faltam na teoria, criaremos aproximações explícitas e verificáveis (com experimentos planejados na seção G) para iterar e refinar conforme evidências.

C) Estado da Arte: Existe Algo Parecido?

Vamos situar nosso projeto frente à literatura existente em áreas relacionadas:

1. **Analogia relacional/estrutural (teorias e engines):** A pesquisa cognitiva e de IA traz várias abordagens de analogia. A **Structure-Mapping Theory (SMT)** de Gentner (1983) já introduzimos: foca em alinhar relações e ignora atributos superficiais [36]. O algoritmo clássico baseado nisso é o **Structure-Mapping Engine (SME)** de Falkenhainer, Forbus & Gentner (1989), que gera mapeamentos possíveis entre dois conjuntos de proposições e avalia pelo princípio da sistematicidade. O SME teve sucesso modelando analogias humanas, mas requer que os conhecimentos dos domínios estejam representados em linguagem lógica estruturada – o que é um gargalo (precisa de entrada *hand-crafted*) [4] [45]. Em contrapartida, houve tentativas de automatizar a obtenção de representações: *Latent Relation Mapping Engine (LRME)* de Turney (2008) combina ideias do SME com análise de corpus textual para descobrir relações semânticas latentes entre palavras [46]. O LRME conseguiu desempenho nível humano em 20 problemas de analogias (10 científicas, 10 metáforas comuns) usando apenas textos brutos como base [5]. Isso sugere que é possível extrair analogias estruturais de dados não estruturados – conceito que inspirará nossos importadores de texto. Outras abordagens incluem o **ACME** (Holyoak & Thagard, 1989), que modelou analogia como um problema de restrição satisfatibilizada via rede neurais (buscando maximizar correspondências e minimizar conflitos simultaneamente), e modelos conexionistas mais recentes como **LISA** (Hummel & Holyoak, ~2005) que combinam representações distribuídas com vinculação temporal para representar estruturas – buscando superar a limitação do SME de representações manuais. Mais recentemente, com deep learning, alguns trabalhos exploram analogia via **aprendizado de representações**: por exemplo, pesquisa em analogia de imagens ou

analogias verbais usando embeddings combinados. Entretanto, **nenhuma solução atual foca explicitamente em analogias multi-domínio complexas** como bio vs software integrando hierarquias, papéis cibernéticos e visual analytics. Nosso enfoque difere por *integrar múltiplas camadas conceituais e quantificar múltiplas dimensões de semelhança simultaneamente*. Ainda assim, beberemos de várias fontes: do SME tomamos a ênfase relacional; do ACME/LISA a ideia de resolver correspondências como otimização global (inspirando nosso algoritmo v2 com ILP); do LRME a ideia de usar corpora textuais para enriquecer representações (importante para preencher nosso grafo automaticamente). Em suma, a literatura nos dá ingredientes, mas *nenhum "engine" existente faz exatamente o que precisamos*, especialmente em ambiente local-first com input heterogêneo.

2. **Ontology Matching e Alignment:** A área de integração de ontologias na Web Semântica lida com problema análogo: dado dois grafos (ontologias) descrevendo domínios similares, encontrar correspondências entre seus termos (classes, propriedades). Ferramentas como **AML (AgreementMakerLight)** e **LogMap** são *state-of-the-art*. Essas ferramentas combinam correspondências lexicais (nomes semelhantes) com estrutura ontológica (posições na hierarquia) e algumas até usam lógica para reparar inconsistências. Em avaliações (como o concurso OAEI), AML e LogMap tipicamente alcançam altos *F-scores* em alinhar ontologias biomédicas [10]. No contexto do nosso projeto, ontologia matching é uma sub-função (especialmente para domínios biologia e talvez para ontologias de arquitetura de software). Podemos aproveitar algoritmos de correspondência lexical sofisticados dessas ferramentas e técnicas de filtragem. Vale notar que sistemas de matching recentes incorporam embeddings e redes neurais: ex. **BERTMap (2021)** que faz fine-tuning de BERT em rótulos de ontologias e supera AML/LogMap em alguns cenários biomédicos [47] [15]. Isso aponta para uso de representações contextuais aprendidas para melhorar correspondências – ideia que podemos aplicar no componente semântico do nosso *hotness*. Contudo, ontologia matching clássico visa relações de equivalência entre conceitos de *mesma ontologia ou similares* (por ex., alinhar "muscle" em uma ontologia com "musculus" em outra – sinônimos). Nós buscamos analogias potencialmente mais abstratas (não necessariamente sinônimos, mas estruturas correspondentes). Ainda assim, técnicas de **graph alignment** e visualização de alinhamentos são relevantes. Um trabalho interessante é de Pesquita et al. (2014) sobre *visualizar alinhamento de ontologias grandes*, propondo interfaces para explorar correspondências em ontologias biomédicas [48] [17]. Eles notam que só listas de pares não bastam – é útil ter visual como grafos ou matrizes. Nossa escolha por heatmap se alinha: Pesquita et al. experimentaram com grafos navegáveis dada a escala. Vamos nos inspirar nas soluções de visual analytics deles, como *foco+contexto* em regiões onde há mais correspondências. Resumindo: ontologia alignment nos fornece algoritmos robustos (que incorporaremos) e lições de UX para não sobrecarregar o usuário com milhares de pares (daí nossa priorização visual de destaques quentes).

3. **Similaridade em grafos (graph similarity):** Problema fundamental em grafos: avaliar quão similares são dois grafos ou subgrafos. Abordagens incluem:

4. **Graph kernels:** funções que computam similaridade baseada em contagem de subestruturas comuns (caminhos, árvores, graflet kernels). Exemplo: *Weisfeiler-Lehman graph kernel* conta padrões de rotulação [49] [50]. Kernels permitem comparar graficamente dois sistemas e até embutir grafos em um espaço vetorial. Porém, muitos kernels focam em isomorfismo aproximado de rótulos e conexões, podendo falhar se os grafos têm mapeamentos parciais complexos (nosso caso).
5. **Graph embeddings (node2vec, etc.):** Geram vetores para nós (ou grafos inteiros) preservando propriedades de adjacência. Por exemplo, *node2vec* faz random walks e aprende vetores para nós de

tal forma que proximidade no vetor indica contexto de vizinhança parecido. Isso pode ajudar a sugerir correspondência nó a nó (nós com vetores próximos podem ser análogos). Já *embeddings de grafos inteiros* (Graph2Vec ou representações via autoencoders) tentam condensar um grafo em vetor – mas dois grafos complexos dificilmente serão próximos se não são quase-isomórficos. De fato, um limite conhecido: muitos GNNs equivalem ao teste de isomorfismo de Weisfeiler-Lehman de 1 dimensão, não capturando diferenças sutis de estrutura [51]. Assim, embeddings podem ajudar na etapa semântica (ex.: comparar significados de nomes ou centralidade) mas para *estrutura relacional rica*, precisaremos casar explicitamente subgrafos.

6. **Subgraph isomorphism:** encontrar um subgrafo de B idêntico a A. Problema NP-completo em geral [52]. Ferramentas como VF2, Ullmann algorithm resolvem para grafos pequenos, mas a complexidade explode. Nosso cenário é mais flexível: permitimos mapeamento parcial com diferenças, não isomorfismo exato – então não podemos usar solvers de isomorfismo diretamente, mas podemos usá-los em subproblemas (ex.: checar se estrutura X aparece dentro de Y).
7. **Algoritmos recentes de matching/alignments:** Além de ontologias, há trabalhos em *graph alignment* via aprendizagem: ex. usar GNNs para aprender a produzir alinhamento ótimo de nós entre dois grafos (visto como sequência de correspondências). Um exemplo: *Graph Alignment Kernel (GAWL)* que aplica refinamento tipo WL para encontrar similaridade alinhando subestruturas [53]. Também frameworks como *CONNA* (2020) combinam embeddings e buscas locais para alinhamento.

Em resumo, ferramentas existem, mas precisamos adaptá-las ao nosso foco: *similaridade funcional*, não apenas estrutural cega. Por exemplo, grafos muito diferentes em tamanho podem ter analogia válida se compartilharem um subgrafo funcional crítico (um laço de controle). Então avaliaremos similaridade via uma combinação: comparação direta de vizinhanças (ideia de kernel) + heurísticas baseadas em tags (ex.: match de “nó sensor” com “nó sensor”). O estado da arte nos alerta para o custo computacional – teremos que ser inteligentes com pruning. Importante: detectamos que **nossos grafos conceituais são rótulados e relativamente pequenos por camada** (não milhões de nós; talvez dezenas ou centenas significativos). Isso torna viável aplicar técnicas exatas localmente. Em casos grandes (um texto enorme virando grafo?), de fato precisaremos limites de profundidade e filtragem semântica para focar.

1. **Visual analytics para matrizes/grafos grandes:** Nossa UI central é um heatmap interativo possivelmente grande. Visualizar matrizes de similaridade ou alinhamento é comum em bioinformática (genomas), em *machine learning* (matriz de confusão) etc. Desafios: desempenho (renderização), legibilidade (cores certas), interatividade (zoom, pan, tooltips). Tecnologias: usar GPU via WebGL ou WebGPU está virando padrão para gráficos intensos. Por exemplo, o pacote *shinyheatmap* atingiu visualização de dataset com 10^5–10^7 linhas em segundos usando plugin de alto desempenho em navegador [54] [55]. Isso foi em 2017; hoje com WebGL podemos similarmente renderizar milhares de células a 60fps, desde que não criemos DOM elements para cada pixel! Técnicas de *Level of Detail (LOD)*: quando o heatmap está *zoomed out*, podemos agrupar células (fazer média ou valor máximo do bloco) – economiza processamento e dá visão macro. Conforme usuário dá zoom ou seleciona região, refinamos. Ferramentas como *HiGlass* (para genomics) fazem isso dinamicamente. Outra técnica importante: *tiling* – pré-calcular tiles de imagem para pedaços do heatmap e carregar sob demanda (similar a mapas web). Adotaremos uma estratégia simples: se matriz > certo tamanho, gerar canvas de resolução limitada e atualizar ao interagir.

No front de interação, *coordenadas ligadas* (brushing & linking) é consagrado: usuário destaca algo numa visual (ex.: uma célula) e em outra visual associada (ex.: grafo textual) vemos highlight correspondente. A pesquisa de visual analytics enfatiza isso para entendimento de dados complexos [25]. Faremos isso com

highlight de nós/arestas correspondentes no painel de explicação quando mouseover na célula do heatmap. Além disso, devemos prover filtros: por camada, por threshold de hotness, busca textual por conceitos (ex.: highlight células envolvendo “apoptose”).

Sobre paletas de cores: a *guerra contra o Rainbow* já foi bem documentada. A escala **jet/rainbow** distorce percepção – por exemplo, amarelo sobressai demais independentemente do valor, verdes podem parecer mesma intensidade quando não são, e é inutilizável para daltonismo vermelho-verde [26]. A comunidade de vis hoje prefere escalas perceptualmente uniformes: **Viridis**, **Magma**, **Plasma** (do matplotlib) são boas candidatas, ou a família **CET** (colorbrewer). Vamos escolher, por exemplo, Viridis (azul->verde->amarelo) ou uma variante que em grayscale ainda seja monotônica. O artigo da Nature (2020) reforça quão crítico é usar escalas científicas para evitar inferências erradas [28] [56]. Em design de ferramentas analíticas, isso não é detalhe – adotaremos colormap adequado desde o MVP.

Em termos de trabalhos específicos: Pesquita 2014 já mencionada sobre visualizar alinhamentos, e outros sistemas de visualização de grafos alinhados (como ferramentas de merges de modelos) priorizam **sumário + detalhamento progressivo**. Em dashboards modernos (Grafana, Kibana etc.), gráficos de calor interativos existem mas geralmente com poucos centenas de pontos. Nosso caso pode ter uns milhares (conceitos de biologia vs conceitos de software). Felizmente isso é administrável hoje com tecnologias de jogos no browser.

1. **Casos bio → software com evidência real:** Existe literatura comparando evolução de software com evolução biológica. Por exemplo, o trabalho de M. Fortuna et al. (PNAS 2011) analisou a rede de pacotes do Debian Linux como um sistema ecológico [16] [17]. Eles observaram aumento de modularidade ao longo do tempo com reutilização de código (análoga a módulos biológicos evolutivos) e notaram que alta modularidade protege contra cascatas de falhas (conflitos entre pacotes ficaram isolados) [48] [17]. Isso dá evidência quantitativa de analogia: software crescendo se comporta como ecossistema onde modularidade = nichos. Outro exemplo: article *“Evolution in the Debian GNU/Linux software network: analogies and differences with gene regulatory networks”* (Royal Society 2014) explorou medidas de rede e comparou com redes gênicas, encontrando padrões similares de distribuição de conectividade. Além disso, blogs e palestras (e.g. *“Evolutionary architecture”* no contexto de microserviços) fazem paralelos com seleção natural, mutações (AB testing), etc. Conway’s Law (1968) relacionando arquitetura de software com estrutura organizacional é outro tipo de analogia que foi validada em estudos (microserviços mapeiam para times ágeis autônomos, semelhante a organismos e ecologias).

No âmbito de *robustez*, analogias: *sistema imune vs segurança cibernética* (conceito de inato/adaptativo vs firewalls estáticos e sistemas de detecção adaptativos) aparecem em pesquisas aplicadas. E conceitos como *degeneracidade* (diferentes partes podem assumir a função de outra se necessário, comum em bio) foram discutidos em software reuso/fault-tolerance.

O ponto é que **há apoio em casos isolados** de que certas analogias bio-software não são só metáforas vagas, mas têm métricas e resultados mensuráveis similares. Citamos o Debian modularidade robustez (Fortuna 2011) como um pilar [48] [17]. Também há quem estude **arquitetura de software via ecologia**: ver *“The Ecology of Software”* (2007) argumentando que softwares competem e se adaptam num ecossistema de tecnologias.

Nosso trabalho sistematizaria isso. Em vez de estudar uma analogia por artigo, teremos ferramenta para explorar *várias analogias simultaneamente e iterativamente*. Isso é novidade.

1. **VSM aplicado a tecnologia/organizações:** Stafford Beer aplicou VSM principalmente a empresas e economia (Cybersyn no Chile). Mas há tentativas de trazer ao desenvolvimento de software. Herring & Kaplan (2008) discutiram *"The Viable System Model for Software"*, propondo interpretar um sistema de software complexo como análogo a uma organização viva [11] [57]. Eles definem "software viability" como a capacidade de evoluir adaptativamente e entrar em modo *intelligent control* [58]. O Medium do Gorelkin (2025) que citamos resume isso: mapeia agentes de IA para subsistemas VSM [37] [38] e discute que desafios de agentes múltiplos (coordenação, exploração vs exploração, manter identidade/políticas) podem ser resolvidos via estrutura VSM [59] [60]. Isso é exatamente a ponte que pensamos: usar *Sistemas 1-5* como camada de comparação abstrata. Então, sim, existe uma conversa corrente sobre VSM e software. Não é mainstream, mas está crescendo no contexto de IA multi-agente e arquiteturas adaptativas. Nosso projeto incorpora essa visão embutindo VSM como "ontologia de papéis" transversal. Assim, contribuiremos também a essa literatura explorando exemplos concretos: e.g., identificar S4 (futuro) de um software – possivelmente módulo de analytics/preditivo – e ver se sistema biológico tem análogo (no organismo, S4 seria cérebro aprendendo ambiente). Alguns artigos práticos: uso de VSM em gerenciamento de serviços de TI (para estruturar equipes DevOps recursivamente), ou aplicação em governança de microserviços (cada microserviço = S1, plataforma = S3, arquitetura geral = S5, etc.). São mais artigos de conferência e blogs do que evidências empíricas rigorosas. Ainda assim, fornecem frameworks que validaremos.

2. **Controle em sistemas computacionais:** Já há bastante aplicação de teoria de controle na computação. Exemplos:

3. *Autoscaling na nuvem:* diversos trabalhos modelam o ajuste de recursos como um controlador feedback que mantém CPU ou latência no setpoint [61] [62]. Controladores PID foram implementados para auto-escalar VMs ou pods, mostrando reduzir overshooting de recursos [63]. Amazon AWS em seus serviços internos usa controladores para estabilidade (ver palestra de Colm MacCárthaigh (AWS) 2020 destacando que control theory é "veia de ouro" para engenheiros de sistemas [64] [65] – ou seja, já aplicam mas nem sempre formalizam).
4. *Controle de congestionamento de rede:* o famoso algoritmo TCP (AIMD) é basicamente um laço de controle distribuído que evita congestionamento. Novos algoritmos (BBR do Google) foram projetados com análise de controle.
5. *Circuit Breakers e padrões resilientes:* São padrões de engenharia de software inspirados implicitamente em controle e biologia. Um **Circuit Breaker** age como um dispositivo de controle on/off para chamadas de serviço: monitora falhas e "abre o circuito" quando um limiar excede, prevenindo chamadas futuras até recuperar [66]. Isso evita cascatas. Fowler e Nygard popularizaram nos anos 2000 [67]. Esse padrão tem analogias claras com *apoptose* (isolar e desligar parte defeituosa) e com *disjuntores elétricos* (daí o nome, inspirado em engenharia elétrica de proteção). **Bulkheads** (Compartimentos): outro padrão, isola recursos por componente (ex.: pools separados), igual compartimentos de navio [68] – analogia explícita naval, mas também lembra "compartimentalização biológica" (organelas isolando processos para que problemas não vazem).
6. *Controlos de fila e backpressure:* sistemas de mensageria usam feedback (ex.: TCP nativo) para desacelerar produtores se consumidores não dão vazão, evitando overflow – um laço negativo clássico.

7. *Sistemas de* auto-tuning*: do nível de SO (controlador de frequência de CPU) a banco de dados autônomos, utilizam controladores adaptativos.
8. *Feature flag* + *canary release*: mecanismos de experimento-controlado e rollback rápido equivalem a ter uma malha de segurança (feedback rápido de falha e ação corretiva – se erro sobe, desligue feature).

Há também literatura que faz ponte explícita: por exemplo, *"Control theory for adaptive software"* (autorizar construções de control loops para gerenciar config de software em runtime) e *"feedback control in cloud"*. Um conceito interessante é **Homeostase em computação**: sistemas que mantêm parâmetros estáveis (Google falou de "homeostatic datacenter cooling").

Resumindo, sim, *muitos conceitos cibernéticos já são aplicados em computação*, mas de forma ad hoc. Nosso projeto reconhece isso e tenta unificar linguagem. Citamos MacCárthaigh/AWS que defende que engenheiros de software deveriam estudar controle porque as soluções estão lá faz décadas [69] [65]. Essa é uma motivação forte para nossa ferramenta: explicitar essas conexões pode evitar reinventar a roda e prevenir instabilidades (quantas vezes times fazem autoscaler e geram oscilações por não pensar como controle?).

Conclusão do estado da arte: Não encontramos um produto ou pesquisa que reúna *ontologia* + *analogia multi-domínio* + *visual analytics* exatamente como propomos. Porém, os campos relacionados nos dão blocos: teoria de analogia para base conceitual, técnicas de ontology alignment para matching, graph similarity para algoritmos, e design de visualização para interface. Vamos montar um *Lego* inovador com essas peças, calibrando conforme nosso objetivo de ferramenta local-first altamente explicativa.

D) Modelo Canônico: Representação de Conceitos em Camadas

Para integrar diferentes inputs (texto, código, ontologias) e sustentar nosso cálculo de analogias, definimos um formato canônico de grafo – chamado aqui de **GraphSpec** – que padroniza a representação multi-camadas. Usaremos JSON (que facilita manipulação em Python/JS) conforme descrito a seguir, seguido de um exemplo mínimo.

Especificação GraphSpec:

- **Node (nó/conceito):** representado por um objeto com propriedades:
- `id`: identificador único canônico (string ou número). Se possível, determinístico (ex.: nome normalizado ou UUID fixo) para permitir merge. Ex: `"bio:Apoptosis"` ou hash do nome.
- `label`: nome legível do conceito. Ex: `"Apoptose"` (português) ou `"Apoptosis"` (decidir consistência de idioma; podemos manter inglês para termos científicos).
- `domain`: domínio ou origem principal do conceito. Ex: `"Biologia"`, `"Software"`, ou `"OrgCyber"`. Pode ser nulo para conceitos muito gerais.
- `layer`: um inteiro indicando a camada/nível hierárquico no qual esse conceito está, relativo ao contexto do seu domínio. Camada 0 = conceito raiz (bem geral), camada 1 = componente imediato, etc. Ex: Apoptose é fenômeno específico dentro de biologia celular, poderia ser camada 1 sob um conceito raiz "Morte Celular".
- `parents`: (opcional) lista de ids de nós pais (conceitos mais gerais dos quais este é subconceito). Permite hierarquia poliparental se aplicável.

- `tags`: lista de tags/categorização. Aqui usamos para papéis cibernéticos, escalas, etc. Ex: `["VSM:S1", "Role:Actuator", "Timescale:Hours", "FailureMode:FailSafe"]`.
- `embedding` (opcional): vetor numérico (e.g. 128-dim) capturando semântica do conceito. Poderá vir de modelo de linguagem ou word2vec treinado. Usado para similaridade semântica rápida (cosine).

- `properties` (opcional): dicionário de propriedades diversas, conforme input. Por exemplo, se veio de ontologia, talvez `{"ontology_id": "GO:0006915", "definition": "Programmed cell death..."}`.

- **Edge (aresta/relação):** representado por objeto:

- `source`: `id` do nó origem.
- `target`: `id` do nó alvo.
- `type`: tipo da relação (string curta). Usaremos um conjunto predefinido, inspirando em ontologias e relações cibernéticas:
    - Hierarquia: `"is_a"` (A *é um* tipo de B), `"part_of"` (A *parte de* B).
    - Causalidade/Influência: `"causes"`, `"regulates"`, `"inhibits"`, `"enhances"`.
    - Sequência/Fluxo: `"precedes"`, `"triggers"`.
    - Comunicação: `"signals_to"`, `"feeds_back_to"` (feedback loop).
    - Dependência em software: `"calls"`, `"reads_from"`, `"writes_to"`.
    - Similaridade/Analogia: (esse tipo a princípio não estará no grafo individual, mas pode surgir no meta-grafo comparativo).
    - Meta-relations (cybernetics roles): `"senses"` (A percebe/sensoreia B), `"acts_on"` (atuador, A age em B), `"setpoint_for"`, `"monitored_by"`.
    - Outros possíveis: `"contrasts_with"` (como Apoptose contrasta com Necrose).
- `weight` (opcional): peso ou força da relação (número). Ex: 0.9 se evidência forte.
- `evidence` (opcional): referência à fonte dessa relação (texto original, documento, linha de código). Pode ser string ou estrutura com `source_id` e trecho. Ex: `{"source": "paperX.pdf", "quote": "Apoptosis is triggered by p53..."}`

- Outras: `directed`: bool indicando se é direcionada (padrão true a não ser que type implicar bidirecional). Para hierarquia, directed (A is_a B). Para *analogias* no meta-grafo, seria não-direcionada.

- **Graph meta:** metadados de todo grafo:

- `id` ou `origin`: identifica a fonte (ex.: nome do arquivo de entrada, ou "Gerado a partir de repositório X").
- `domain`: se o grafo for monodomain (ex.: inteiramente Biologia), especificar.
- `generated_at`: timestamp.
- `version`: se for derivado de versão de código ou ontologia versionada.
- `confidence`: se quisermos dar uma noção de confiança da extração (ex.: texto não estruturado vs ontologia confiável).

Camadas e subconceitos: Representamos explicitamente via `parents` e tags de camada, mas para conveniência, ao gerar GraphSpec de certos formatos (ex.: código, que naturalmente tem estrutura pasta -> módulo -> função), podemos aninhar na estrutura JSON para debug/hierarquia visual. Contudo,

internamente trataremos sempre como um grafo plano com arestas `is_a` ou `part_of` conectando camadas. Assim evitamos explosão combinatória: se um conceito tem 50 subconceitos, eles serão nós filhos ligados por `part_of`, e não replicaremos todas as combinações. Além disso, **limitaremos profundidade de extração**: por padrão, não passar camada 2 ou 3 a menos que explicitamente pedido (evitar que um texto grande gere grafo gigantesco). Também **filtraremos por relevância**: ex.: se do texto extraímos 100 frases mas apenas 10 parecem definidoras do conceito, guardamos só essas relações. Podemos usar TF-IDF ou frequências para selecionar subconceitos importantes.

Consistência entre fontes: se importamos múltiplas fontes (ex.: uma ontologia e um texto) que mencionam o mesmo conceito, idealmente mesclar os nós. Faremos **normalização de labels** (remoção de plural, lowercasing, substituição de sinônimos conhecidos). Também podemos ter um dicionário de mapeamento (ex.: "cell death" -> "apoptosis" se há correspondência exata). Quando não tivermos certeza, talvez mantenhamos nós separados com uma relação `"possible_same_as"` e deixemos o alinhamento para a fase de analogias (ex.: nosso próprio algoritmo pode descobrir que "programmed cell death" do texto = "Apoptosis" da ontologia). Um cuidado: identificadores de código (camelCase etc.) precisam ser divididos e normalizados.

Exemplo mínimo (GraphSpec JSON): Vamos construir um exemplo simplificado com dois domínios – trecho do grafo de Biologia e de Software – e mostrar possíveis nós e arestas:


```
{
  "id": "ExampleGraph_BioSoftware",
  "generated_at": "2026-01-17T21:20:00Z",
  "graphs": [
    {
      "domain": "Biologia",
      "nodes": [
        {
          "id": "BIO_APOPTOSIS",
          "label": "Apoptosis",
          "layer": 0,
          "parents": [],
          "tags": ["Role:Process", "FailureMode:FailSafe", "VSM:S1"],
          "embedding": [0.12, -0.08, ...]
        },
        {
          "id": "BIO_CELL_DAMAGE",
          "label": "Cellular Stress",
          "layer": 1,
          "parents": ["BIO_APOPTOSIS"],
          "tags": ["Role:Trigger"],
          "embedding": [0.05, 0.10, ...]
        },
        {
          "id": "BIO_CASPASE",
          "label": "Caspase Cascade",
```

```
          "layer": 1,
          "parents": ["BIO_APOPTOSIS"],
          "tags": ["Role:Mechanism"],
          "embedding": [0.02, -0.01, ...]
        }
      ],
      "edges": [
        { "source": "BIO_CELL_DAMAGE", "target": "BIO_APOPTOSIS", "type":
"triggers" },
        { "source": "BIO_CASPASE", "target": "BIO_APOPTOSIS", "type":
"part_of" },
        { "source": "BIO_APOPTOSIS", "target": "BIO_CELL_DAMAGE", "type":
"regulated_by", "evidence": "p53 sensing DNA damage" }
      ]
    },
    {
      "domain": "Software",
      "nodes": [
        {
          "id": "SW_CIRCUIT_BREAKER",
          "label": "Circuit Breaker",
          "layer": 0,
          "parents": [],
          "tags": ["Role:Component", "FailureMode:FailSafe", "VSM:S1"],
          "embedding": [0.10, -0.05, ...]
        },
        {
          "id": "SW_ERROR_RATE",
          "label": "Error Counter",
          "layer": 1,
          "parents": ["SW_CIRCUIT_BREAKER"],
          "tags": ["Role:Sensor"],
          "embedding": [0.07, 0.09, ...]
        },
        {
          "id": "SW_RETRY_LOGIC",
          "label": "Retry Logic",
          "layer": 1,
          "parents": ["SW_CIRCUIT_BREAKER"],
          "tags": ["Role:Mechanism"],
          "embedding": [0.00, 0.03, ...]
        }
      ],
      "edges": [
        { "source": "SW_ERROR_RATE", "target": "SW_CIRCUIT_BREAKER", "type":
"monitors" },
        { "source": "SW_RETRY_LOGIC", "target": "SW_CIRCUIT_BREAKER", "type":
"part_of" },
```

```
        { "source": "SW_CIRCUIT_BREAKER", "target": "SW_ERROR_RATE", "type":
"acts_on", "evidence": "Trips open when error_rate > threshold" }
      ]
    }
  ]
}
```

(Exemplo ilustrativo; embedding vetoriais truncados; evidence textual simplificada.)

Nesse exemplo, **Apoptosis** (BIO_APOPTOSIS) tem subconceitos *Cellular Stress* e *Caspase Cascade*, ligados por `triggers` e `part_of`. **Circuit Breaker** (SW_CIRCUIT_BREAKER) tem *Error Counter* (sensor de falhas) e *Retry Logic*, com relações análogas. Note que ambos Apoptosis e Circuit Breaker compartilham tags *FailureMode:FailSafe* e papel S1 (são componentes operacionais do sistema maior). Esse GraphSpec segmenta os dois grafos por domínio, mas numa implementação real podemos unificar num grafo só – aqui separamos didaticamente. O importante: todos conceitos-chaves estão estruturados, prontos para o algoritmo alinhar **Apoptosis ⇄ Circuit Breaker**, **Cellular Stress ⇄ Error Counter** (ambos "gatilhos"), etc., e calcular *hotness* dessa analogia.

Extração de subconceitos (importers): - Para **texto**, imagina-se usar processamento de linguagem: extrair entidades e relações (via ferramenta de NLP como spaCy, ontologies DBpedia, ou mesmo um modelo transformer QA invertido para extrair triplas). Provavelmente combinamos: detectar termos chave (substantivos) e verbos relacionando-os. Por exemplo, de um artigo: "Apoptosis is triggered by cellular stress via p53" extrair ("Cellular Stress" – triggers → "Apoptosis"). Faremos isso recursivamente para construir subárvore. - Para **repositório de código (Python)**, extrairemos: componentes (classes, funções → nós layer 0/1), chamadas entre funções (-> edges "calls"), dependências de módulo (-> edges "imports" que podem indicar arquitetura), e possivelmente padrões de projeto se detectáveis (ex.: uso de circuit breaker library -> tag). - Para **ontologias (OWL/RDF)**, basta percorrer classes e propriedades, criando nós e edges equivalentes. Ex.: se ontologia diz *Apoptosis is_a CellDeath*, criaremos BIO_APOPTOSIS node com parent de id de CellDeath. - **Mistura:** se temos duplicatas, como mencionado, normalização e mescla. Podemos priorizar ontologia para taxonomia e texto para causas/efeitos narrativos.

Controle de complexidade: Implementaremos limites configuráveis: `max_layers` (profundidade), `max_children_per_node` (top-K subconceitos por conceito, se muitos encontrados). Como priorizar quais K? Podemos ordenar por frequência (termo mais mencionado no texto), por importância no ontologia (classes definidas vs inferidas), ou manual (usuário marca quais acha relevantes). Um futuro pode envolver UI para o usuário podar/expandir nós do grafo conforme veja fit.

Consistência de labels/IDs: Manteremos um registro global de nomes já vistos. Por exemplo, se texto traz "programmed cell death" e ontologia tem "apoptosis", detectamos alta similaridade (mesmo embedding ou via um dicionário de sinônimos) e unimos. A GraphSpec não proíbe duplicatas, mas preferimos mesclar porque nossa analogia opera em correspondências entre domínios, não duplicatas dentro do mesmo. Mas se duplicatas passarem, o algoritmo de matching ainda pode lidar (acabará mapeando ambos similares talvez para o mesmo correspondentes do outro lado – teremos então que decidir um merge tardio).

Em resumo, o GraphSpec serve como **linguagem pivô**: qualquer fonte entra, é convertida a GraphSpec (nós + arestas com tipos padronizados), e então algoritmos e visualização trabalham sobre esse formato

unificado. Isso garante extensibilidade – amanhã podemos adicionar importer de documentação (ex.: um Markdown de ADR – *architectural decision record*) extraindo decisões e ligando a componentes, e tudo se encaixa no grafo.

E) Hotness Score: Como Medir o "Calor" de uma Analogia

Definiremos o *hotness* como um escore calculado a partir de várias dimensões complementares, conforme introduzido na seção B. Aqui detalhamos cada dimensão, como computá-la e combiná-las, além de como explicar o resultado por camada.

Dimensões de Similaridade e suas Métricas:

1. **Semântica:** Avalia quão semelhantes são os conceitos alinhados em termos de significado linguístico ou contextual. Métodos:
2. Se nós têm embeddings (vetores) pré-calculados, usar a similaridade coseno entre vetores do nó A (no domínio-fonte) e nó B (no domínio-alvo). Ex: coseno > 0.8 sugere alinhamento forte de significado. Esses embeddings podem vir de modelos tipo Word2Vec, GloVe ou BERT (média de tokens).
3. Fallback lexical: se não houver embedding, usar distância de string normalizada (Jaro-Winkler, trigram) para capturar casos de nomes parecidos (e.g. "Apoptosis" vs "Apoptose" ou "Cell" vs "Celula").
4. No caso de conceitos compostos, podemos decompor e comparar subconceito a subconceito.
5. Obter um escore S_{sem} entre 0 e 1.

Racional: Mesmo que analogia não dependa de palavras iguais (pelo contrário, analogias boas às vezes têm palavras bem diferentes, ex.: *hydrogen atom vs solar system*), a similaridade semântica alta pode indicar correspondência trivial (literal similarity) [70]. Portanto, S_{sem} é uma dimensão que não é obrigatória para analogia, mas é informativa – e se for alta, pode sugerir que a "analogia" é na verdade só duas visões do mesmo conceito (o que não é negativo, mas menos interessante). No composto final, esse peso pode ser menor que os demais, para não "recompensar" analogias óbvias demais.

1. **Estrutural/Relacional:** Mede quanto da **estrutura de relações** em torno do conceito de um lado existe de forma correspondente do outro lado. Essencialmente, comparamos subgrafos:
2. Para um par de conceitos propostos (A em domínio1, A' em domínio2), consideramos seus vizinhos (pais, filhos, conexões causais, etc.). Precisamos encontrar um mapeamento entre vizinhos de A e vizinhos de A' que maximize match de tipos de relação.
3. Podemos simplificar: computar *assinaturas* do nó e comparar. Exemplo de assinatura: lista de triplas [rel_type, rel_dir, neighbor_tag]. Para Apoptose: [(triggers, out, CellularStress), (part_of, in, CellDeath), ...]. Para CircuitBreaker: [(monitors, in, ErrorRate), (acts_on, out, Service)]. Claro, nomes diferentes – mas olhamos padrões: Apoptose tem uma causa e faz parte de um processo maior; CircuitBreaker é acionado por algo e faz parte de um sistema maior? Se sim, similar.
4. Poderíamos usar um **graph kernel local**: ex.: contar quantas relações do tipo "A causa X" correspondem a "B causa Y" com X análogo a Y. Ou quantas estruturas triádicas (A->X, A->Y) replicam (B->X', B->Y') com X' análogo a X etc.
5. Talvez mais simples: montar uma pequena matriz de adjacência local e usar Hungarian algorithm para casar vizinhos. Por exemplo, Apoptose vizinhos = {CellularStress (trigger), CaspaseCascade (mechanism)}; CircuitBreaker vizinhos = {ErrorCounter (sensor), RetryLogic (mechanism)}. Casar

CaspaseCascade–RetryLogic (ambos mechanism), CellularStress–ErrorCounter (trigger vs sensor, não igual mas ambos são entrada; meio conflitante, porém ambos antecedem o evento principal). Podemos atribuir pontuações: mesma relação e tags => +1, relação diferente ou tag diferente => penalização.

6. Também verificar preservação de *estrutura global*: se Apoptose regula Necrose (hipotético), e CircuitBreaker regula "CatastrophicFailure" de sistema, isso seria uma correspondência estrutural maior (2 passos).
7. Resultado: uma medida S_{str} de 0 a 1, representando proporção de relações que conseguiram mapear bem. Fórmula inicial: $S_{\text{str}} = \frac{2m}{R_A + R_B}$, onde m = número de relações mapeadas mutuamente, R_A = nº de relações relevantes de A, R_B = de B. (Similar à métrica de F-score estrutural).

8. Além da quantidade, consideramos *ordem/nível*: analogias boas frequentemente mapeiam relações de ordem superior (relacionamentos entre relacionamentos) [31]. Por ora, manteremos 1ª ordem (diretas), mas nosso design multi-layer já traz hierarquia (pais/filhos).

9. **Funcional/Role match:** Verifica se os elementos comparados cumprem funções correspondentes nos seus sistemas, principalmente usando as **tags de papel cibernético e VSM**.

10. Exemplo: Apoptose tem tag `FailureMode:FailSafe`, CircuitBreaker também, ótimo. Apoptose tem `Role:Process` de remoção, CircuitBreaker `Role:Component` de proteção – não exatamente igual, mas ambos atuam como *mecanismo de proteção*, então possivelmente sim. Se ambas têm tag VSM:S1 (unidades operacionais), ponto positivo.
11. Formalmente, podemos ter um vetor binário de categorias para cada nó (tags normalizadas como features) e computar similaridade (ex.: Jaccard ou coseno se tratarmos como vetor).
12. Talvez mais eficaz: conjugar com estrutura: Apoptose está inserida num loop de feedback? (ex.: detecta dano -> executa -> sinaliza para fagócitos). CircuitBreaker insere num loop? (monitor->tripa->reset). Se ambos formam loops de controle completos, é um match funcional topológico.
13. Score S_{func} dado por combinação: similaridade de tags + similaridade de posição no VSM. Ex: se A é S1 e B é S1 (ambos operações de linha) ou A é S4 e B é S4 (ambos planejamento), +1; se não, 0.5 se ambos roles técnicos combinam (ex.: sensor vs sensor).

14. Vale incorporar *Requisite Variety* aqui: se sistema A e B têm variedade comparável ou se as funções se alinham em fechar variedade. Ex: Apoptose resolve falhas imprevisíveis internamente (variedade de falhas celulares), CircuitBreaker lida com variedade de falhas de serviços. Ambos atuam na fronteira componente-sistema para impedir spread – papéis equivalentes de *absorver variedade* (Ashby). Como quantificar? Poderíamos ter meta-info do tipo "variedade distúrbios que este mecanismo cobre" e comparar ratio. De início, qualitativo: se ambos explicitamente fail-safe, supomos função análoga.

15. **Timescale e Failure-mode alignment:** Dimensão que checa se os sistemas operam em escalas comparáveis e se lidam com falhas de natureza similar.

16. *Timescale:* Cada nó ou relação pode ter atributo de tempo (ex.: "Apoptosis ocorre em horas" vs "Circuit breaker atua em segundos"). Se um fenômeno é muito mais lento ou contínuo que outro, analogia pode esfriar. Por outro lado, escalas não precisam bater exatamente se o padrão subjacente é escalável (fractal). Mas em geral, um match melhor quando escalas não divergem

demais. Podemos definir categorias (real-time, seconds; adaptive, minutes-hours; evolutionary, meses-anos) e ver se ambos caem na mesma.

17. *Failure mode:* classificar se a falha endereçada é abrupta vs gradual, interna vs externa. Apoptose lida com falha interna (dano DNA, etc.), CircuitBreaker com falha externa (serviço vizinho mal). Ainda assim, ambos previnem cascata – direções opostas (in vs out). Podemos pontuar parcialmente.

18. Então S_{scale} podia ser média de duas sub-notas: S_{time} e S_{failmode}. Ex.: time concorda (ambos dinâmicos de runtime) = 1, failmode concorda (ambos fail-safe preemptivos) = 1; média =1.

19. **Generatividade (opcional):** Mais subjetivo: se a analogia sugere *novas perguntas ou soluções*. Isso poderíamos derivar do subgrafo mapeado: quanto mais correspondências conseguimos, mais inferências podemos tentar. Ex.: se Apoptose ↔ CircuitBreaker, e sabemos Necrose é contraparte ruim de Apoptose, poderíamos inferir "o que seria Necrose em software?" – se a ferramenta descobre ou sugere isso (talvez "Failing unisolated: cascading failure"), analogia é fértil. Podemos medir generatividade pelo número de *novos mapeamentos e hipóteses* que surgem além dos fornecidos. Inicialmente, esta dimensão ficará qualitativa e fora do escore composto automático (pode entrar mais tarde com feedback do usuário marcando "aprendi algo novo com esta analogia").

Score Composto: Podemos combinar as dimensões (1)-(4) via média ponderada:

$$\text{Hotness} = w_{\text{sem}} S_{\text{sem}} + w_{\text{str}} S_{\text{str}} + w_{\text{func}} S_{\text{func}} + w_{\text{scale}} S_{\text{scale}}.$$

Inicialmente, $w_i = 1$ igualmente (ou 0.25 cada, normalizando em 0–1). Ajustes: se certos tipos de analogia importam mais relações e menos semântica, podemos diminuir w_{sem}. Poderemos calibrar pesos via feedback de especialistas: mostrar analogias e perguntar qual foi mais útil, então ajustar pesos para replicar esse ranking (um tipo de aprendizado de preferência).

Explicação do Score: Não basta cuspir número; para cada correspondência de conceitos (célula no heatmap), forneceremos breakdown: - Similaridade semântica: ex. "0.15 (baixo) – termos diferentes, sem parentesco léxico." - Estrutural: "0.9 – 9 de 10 relações de Apoptose encontram paralelo em CircuitBreaker (ex.: *trigger* vs *monitor*, *cascade* vs *retry*)." - Funcional: "0.8 – ambos atuam como fail-safe S1 components; Apoptose envolve sensor p53, Circuit Breaker tem sensor de erro (paralelo claro)." - Escala: "0.7 – Apoptose ocorre em horas, CB em segundos (diferem um pouco), mas ambos atuam antes da falha catastrófica (mesmo tipo de prevenção)." - Com isso, a ferramenta gera frase do tipo: "**Hotness 0.80** = alto devido a forte correspondência estrutural e funcional; semelhança semântica baixa (sistemas diferentes), escala parcialmente alinhada (um mais lento)."

No painel, destacaremos os **top matches de subconceitos** e relações: ex.: Apoptose:CasB vs CB:Retry (match mecânico), Apoptose:DNA damage vs CB:ErrorCounter (match de gatilho) – enumerar talvez os 3 melhores e 3 lacunas (onde falhou mapear). Essas divergências são importantes ("quinas"): ex.: Apoptose tem implicação de *clearance by phagocytes*, software não tem análogo (isso pode ser analogia faltante: haverá componente de "garbage collection"? Se não, apontar: "Software lacks explicit cleanup mechanism akin to phagocytes – possível ponto fraco").

Cálculo por camada: Vamos definir hotness em níveis: Hot_0 comparando conceitos de camada 0 (macroestrutura), Hot_1 comparando subcomponentes imediatos, etc. - *Hot_0* é meio nosso score geral se

considerarmos só as entidades principais. - *Hot_1* refinaria: por exemplo Apoptose vs CircuitBreaker, Hot_0 = 0.8. Mas se olharmos camada 1 correspondências, podemos computar sub-escores para *gatilho*, *mecanismo* etc. Talvez valha como: Hot_d = média de hotness das correspondências de camada d alinhadas. E um *decaimento* para agregação: analogia profunda boa vai ter bons Hot_0, Hot_1, Hot_2; se degrade rápido (Hot_0 alto mas Hot_2 zero), sinal de "quinas" escondidas. - Poderíamos definir $H_{\text{agg}} = \frac{\sum_{d=0}^D \alpha^d H_d}{\sum_{d=0}^D \alpha^d}$ com α um fator de decaimento (ex. 0.8). Assim, camadas básicas pesam mais mas camadas adicionais influenciam. Isso encaixa com *sistematicity principle* – valorizar estruturas profundas conectadas. - No relatório ao usuário, podemos mostrar um gráfico de hotness vs camada: "Camada0: 0.9, Camada1: 0.8, Camada2: 0.4" e dizer: "Quente nos conceitos centrais, mas divergências surgem nos detalhes de camada 2." Isso já orienta se a analogia é útil para big picture mas não pormenores.

Incerteza e falso quente: Analogias podem marcar alto erroneamente – ex.: pura coincidência estrutural sem causalidade real. Como evitar? Uma abordagem: atribuir intervalos de confiança via *bootstrap* – se os embeddings têm variância, se múltiplos possíveis matchings estruturais existem, podemos ver a distribuição. Talvez overkill. Mais simples: correlacionar as dimensões – se só semântica é alta e todo resto baixo, ou só estrutural muito alto mas funcional zero, suspeito. Podemos marcar "possível analogia espúria" se o perfil for estranho. Também integrando plausibilidade de analogia via conteúdo: ex. se causalidades mapeadas envolvem fenômenos incompatíveis (tipo analogia "Sol–Hidrogênio atomo" é boa estrutural, mas se alguém tentasse mapear "Café–Sistema Solar" teria nenhuma base causal ou funcional, seria anomalia [71]). Implementaremos checagens: *Systematicity alone?* (ver refutação de Gentner estrita) – se pontuação estrutural altíssima mas semântica e funcional ~0, levantar flag "pode ser coincidência estrutural" [72] [3].

Em última análise, o *hotness* serve para rankear e chamar atenção; não substitui avaliação humana. Por isso, incluiremos na explicação advertências se necessário: ex. "Analogia parece quente estruturalmente, mas cuidado: os papéis diferem (risco de analogia imprópria)." Daremos ferramentas para o usuário julgar, mantendo honestidade sobre limites (ver seção K).

F) Algoritmos: Como Alinhar Subconceitos (Sem Ficar Caro Demais)

Projetamos três níveis de complexidade para os algoritmos de alinhamento de grafos conceituais, correspondendo às versões evolutivas da ferramenta (MVP, v2, v3). Em cada nível, discutimos abordagem, custo computacional, qualidade esperada e mecanismos de otimização (cache, incremental).

Nível 1 – MVP (Greedy + Heurísticas de Vizinhança):

- *Abordagem:* Iniciaremos com um algoritmo guloso (greedy) que constrói alinhamentos incrementais. Funciona assim: para cada nó no grafo do domínio A, tentamos encontrar o melhor nó correspondente no grafo do domínio B, considerando primeiramente similaridade semântica e tags. Ordenamos pares candidato por uma pontuação inicial (e.g. coseno do embedding + bônus se compartilham tags VSM). Então, aplicamos restrições: evitamos mapear um mesmo nó B para dois nós A diferentes (injeção). Começamos pelos pares de maior score e fixamos como correspondência, então removemos esses nós de consideração para outras.
- *Matching local de vizinhança:* Ao mapear um par, também consideramos seus vizinhos: se A->X e B->Y com A-B já mapeados, isso sugere X-Y como correspondência potencial (estrutura). Podemos incrementar scores de pares (X,Y) candidatos se seus vizinhos estão se alinhando. Essa técnica estilo

"Simulated Annealing" leve ou *iterative deepening* do match: primeiro passe sem estrutura, segundo passe refinando com estrutura.

- *Qualidade esperada:* Esse método é rápido (O(N log N) para ordenar similaridades, com N = nº de nós) e deve pegar correspondências óbvias (mesmo papel, contexto similar). Pode perder alinhamentos globais se, por exemplo, duas correspondências fortes localmente impedem uma configuração melhor envolvendo troca (problema de ótica local vs global). Mas para MVP, é aceitável; teremos validação humana no loop.
- *Custo computacional:* Com embeddings precomputados, comparar todos nós A vs todos B é O(|A|*|B| * dimEmbed). Se 100x100, tranquilo. Mesmo 1000x1000 = 1e6 comparações – borderline mas com otimizações (podar por threshold ou KD-tree de embeddings) é viável local. A parte de vizinhança é O(E) ~ linear nos relacionamentos, também administrável.
- *Cache e incremental:* Como local-first, podemos guardar o resultado do alinhamento A-B. Se um grafo muda ligeiramente (nova versão do repo), recalcular do zero seria ok se N pequeno, mas se N grande, possivelmente precisamos *delta update*: Comparar grafo novo vs antigo, identificar nós novos ou alterados, rodar matching apenas para esses. Podemos projetar IDs estáveis para nós (ex.: "Function X v2" mesmo id de v1), assim correspondências antigas permanecem válidas e só reavaliamos entorno modificado.

Nível 2 – v2 (Matching Ótimo com Constraints):

- *Abordagem:* Modelamos o alinhamento como problema de otimização global. Podemos utilizar o **Algoritmo Húngaro** para resolver maximização de peso de correspondências um-para-um. Primeiro, precisamos de uma matriz de similaridade entre todos pares (A_i, B_j). Essa "matriz de custo" pode ser 1 - (score sem+estrut) ou similar. Aplicando Hungarian, obteríamos um alinhamento bijetivo de cardinalidade = min(|A|,|B|) maximizando soma dos scores. Isso dá correspondência globalmente ótima segundo o critério definido.
- *Restrições Relacionais:* Real analogical mapping exige consistência: se A_i -> B_j e A_i tinha relação R com A_k, idealmente B_j deve ter relação comparável com B_x que corresponde a A_k. Podemos incorporar isso adicionando penalidades de custo: se no matching proposto, certos pares não respeitam estrutura, diminuir peso. Isso torna o problema não-linear (interdependente). Para lidar, podemos iterativamente ajustar a matriz de custo: comece com similaridade semântica, depois de uma solução inicial, penalize pares inconsistentes e resolve de novo. Alternativamente, formular como **ILP (Integer Linear Program)**: variáveis x_{i,j} = 1 se A_i->B_j. Adicionar restrições do tipo: x_{i,j} + x_{k,l} - 1 <= y_{(i,k),(j,l)} (representando se i->j e k->l estão selecionados, então preferimos y=1 se relação condiz). Isso fica complexo rapidamente (muitas restrições).
- *Estratégia prática:* Poderíamos limitar correspondências candidatas: usar top-K candidatos para cada nó (K pequeno, digamos 3). Isso torna ILP manejável pois número de pares possíveis cai. Usar solvers MILP (CBC, Gurobi) localmente para K=3 e N=100 pode ser factível.
- *Custo computacional:* Hungarian é O(n^3) para n = 100 ~ 1e6 ops, ok. ILP depende restrições, mas para N até ~50-100 pode resolver rápido se solver robusto. Ainda, podemos dividir por camada: alinhar primeiro camada 0 (que tende a ter poucos conceitos raiz), depois dentro de cada subparte, resolver alinhamento local para subgrafos (dividir conquista).
- *Qualidade esperada:* Esse método deve aumentar precisão de correspondências complexas, evitando erros como dois nós de A mapeados pro mesmo B. Também deve melhorar recall de pares não triviais mediante relações (descobre correspondência porque fecha lacuna de estrutura).
- *Cache e incremental:* O ILP ou Hungarian completo talvez precise reexecutar se muita mudança. Mas poderíamos travar correspondências confirmadas (ex.: user accepted) como constantes e rodar

otimização só para o restante, assim não redistribui tudo se não necessário (ou se redistribuir, pelo menos manter as travadas).

Nível 3 – v3 (Aprendizado de Máquina Estrutural):

- *Abordagem:* Aqui incorporamos modelos de aprendizado para melhorar ranking de correspondências e talvez aprender a combinar as dimensões de score de forma não linear. Duas vertentes:
- **GNN (Graph Neural Network):** Treinar um GNN que recebe os dois grafos (ou suas partes) e produz um embedding combinado ou até predição direta de correspondência (um tipo de Siamese network ou graph matching network). Por exemplo, um modelo inspirado em *GMN* (Graph Matching Networks) que itera mensagens entre grafos tentando alinhar nós. O GNN poderia ser treinado em correspondências conhecidas (ontologias alinhadas) para capturar padrões gerais.
- **Metric Learning:** Aprender uma função de similaridade que pesa as dimensões de modo ótimo. Por exemplo, um pequeno MLP que entra [sem_sim, struct_sim, func_sim, scale_sim] e foi treinado para output 1 se analogia válida, 0 se não, com base em dataset de analogias rotuladas ou geradas.
- **Node embeddings contextuais:** Poderíamos gerar um embedding para cada nó que incorpora a estrutura (ex.: executar algumas rodadas de message passing GNN para incorporar vizinhos e tags), e então usar coseno desses embeddings contextuais como similaridade final.
- *Custo:* Um GNN rodando local em grafos de poucas centenas de nós seria rápido (milissegundos). O mais custoso é treinar – mas podemos treinar offline ou incremental com feedback do usuário (reinforcement learning light: ajustar pesos se usuário disse analogia X não faz sentido).
- *Qualidade:* Com dados suficientes, o ML poderia captar sutilezas (padrões frequentes de analogia) e generalizar para novas. Ex.: aprender que "sempre mapear controlador→controlador yields melhor outcome do que controlador→sensor" e assim priorizar esses. Também pode calibrar pesos de semântica vs estrutura caso a configuração manual não seja ótima. Contudo, risco de overfitting se dataset pequeno. Precisaríamos possivelmente usar ontologias existentes (biomedical alignments) e fabricar analogias sintéticas (pegar uma ontologia e perturbar).
- *Reranking por constraints:* Mesmo com ML, manteríamos uma etapa determinística final para garantir consistência (por exemplo, usar outputs do modelo como pesos e rodar Hungarian, combinando melhor de ambos mundos – chamado *"learning to match"*).
- *Incremental:* ML permite *inference caching* – se grafos não mudam muito, embeddings GNN de nós já calculados podem ser armazenados e apenas recalcular localmente. GNNs também podem ser atualizados parcialmente (fazer fine-tune em novos correspondências confirmadas sem re-treinar do zero).
- *É viável local?* Sim, bibliotecas de ML leves (Torch CPU, ONNX runtime) podem rodar local sem GPU para grafos pequenos. GNN do tipo GraphSAGE ou GCN com 2-3 camadas seria rápido. O treinamento pode ser mais pesado, talvez feito pelo usuário opcionalmente ou fornecido pre-treinado.

Incremental e Local-First Considerações: - Qualquer nível, manteremos correspondências calculadas e escores em armazenamento local (SQLite DB). Assim, em próxima execução, se os dois grafos não mudaram, nem calculamos de novo – só recuperamos do cache (poderíamos versionar por hash do input). - Para grande escala, permitir alinhamento *on-demand*: se usuário seleciona um subconceito e pede analogias, calculamos centrado nele (subgrafo isomorphism search). - Locally, sem servidor de GPU, a v3 deve ser opcional – e se usar, ter fallback. Provavelmente, v2 com ILP já chega muito longe para 90% casos, e v3 refinaria.

Resumo das Fases: - *MVP:* Implementar similaridade semântica + heuristicazinha estrutural, rankear correspondências, mostrar no heatmap. Isso dá funcionalidade básica. - *v2:* Garantir global optimalidade (menos duplicatas ou perdas) e introduzir penalizações de incoerência – aumenta confiabilidade e "sem quinas". - *v3:* Aprender pesos e padrões a partir de feedback/dados – ferramenta fica mais inteligente, possivelmente sugerindo analogias não óbvias automaticamente.

Nosso design permite escalar esforço: MVP já útil para usuário expert ajustar mentalmente. Versões posteriores automatizam mais, preservando correção.

G) Validação: Como Saber se o Heatmap Faz Sentido?

Implementar e calcular é bom, mas precisamos provar (para nós e usuários) que os resultados refletem algo real e útil, e detectar erros/viés. Propomos um plano multifacetado:

1) Ground Truth (Verdade de referência):

- **Validação Humana Especialista:** Montar um conjunto de pares de conceitos (bio vs software) que especialistas concordam serem analógicos e um conjunto de pares que são analogias ruins. Por exemplo, podemos pedir a um grupo de 5 especialistas (biólogo, engenheiro de software, etc.) para avaliar, numa escala, analogias como "Apoptose ~ Circuit Breaker", "Sistema Imune ~ Sistema de Segurança", "Mutação genética ~ Bug de software" etc. Consolidar casos positivos (analogia válida) e negativos (metáfora furada). Isso servirá tanto para calibrar thresholds de hotness quanto para avaliar se nosso ranking pega os positivos no topo e deixa negativos embaixo. Podemos medir **precisão@k** – dos top 10 analogias sugeridas pela ferramenta, quantas estão no conjunto aprovado? E **recall** – dentre as analogias esperadas, quantas a ferramenta conseguiu detectar e destacar?
- **Corpora e Benchmarks existentes:** Aproveitar ontologias alinhadas como OAEI Anatomy (que alinha parte de anatomia humano vs rato). Embora não seja analogia entre domínios radicalmente diferentes, serve para testar o algoritmo de matching estrutural e semântico: se nossa ferramenta roda nesses ontologias e recupera >90% dos alinhamentos conhecidos (que AML/LogMap já resolvem), sabemos que a base do matching é sólida. Além disso, podemos usar WordNet ou corpus de analogias verbais (SAT analogies: A:B :: C:D) para ver se nosso pipeline consegue posicionar corretamente.
- **Casos clássicos documentados:** Por exemplo, analogia Rutherford (átomo ~ sistema solar). Nosso input poderia ser pequena ontologia do átomo e do sistema solar e ver se hotness sai alto. Ou analogia do *computador* como *cérebro* (McCulloch & Pitts); ou *empresa* como *organismo*. Esses casos foram discutidos em literatura – se nosso sistema concordar com autores (ou explicar divergências), é uma validação qualitativa.

2) Métricas Quantitativas de Desempenho:

- **Precision e Recall de alinhamentos:** Para um ground truth de correspondências (como OAEI ou lista de analogias esperadas), calcular: Precision = acertos/(acertos+falsos positivos); Recall = acertos/(acertos+falsos negativos). Por exemplo, se sabemos 10 pares de conceitos que *deveriam* mapear e nosso sistema achou 8 corretamente (2 perdidos) e sugeriu 2 extras errados, Precision = 8/10=0.8, Recall = 8/10=0.8. Queremos altos, pelo menos comparáveis a ferramentas de ontology

matching top (que muitas vezes têm 0.95+ em conjuntos bem definidos, mas no nosso caso analogias são mais abertas, então 0.8-0.9 seria bom).

- **Consistência por camada:** Definir uma métrica que avalie se anomalias nas correspondências aumentam em camadas profundas. Talvez medir correlação entre hotness de camada 0 e médias das camadas seguintes. Em analogias boas, deve decrescer suavemente, não despencar. Estatisticamente, podemos avaliar nosso conjunto de analogias confirmadas: calcular Hot_0, Hot_1, Hot_2..., e ver se padrão condiz com esperado (p.ex. analogias válidas: Hot_0 > 0.7 e não caem abaixo de 0.3 até camada 2; analogias ruins: Hot_0 alto mas Hot_1 ~0 – band-aid).
- **Estabilidade temporal (versões):** Tomar um sistema de software em duas versões (v1 monólito, v2 microservices) e comparar analogias com biologia antes e depois. Esperamos que se software ficou "mais modular como biologia", o hotness com analogias modulares (ecossistema, organismo multicelular) suba. Além disso, se pequenas mudanças no input ocorrem, idealmente a saída muda localmente mas mantém correspondências globais. Podemos simular: pegar nosso grafo e adicionar um nó irrelevante e ver se de repente heatmap todo muda (não deveria). Isso indica robustez. Métrica: *diferença média nos scores* antes/depois da mudança vs *tamanho da mudança no grafo*. Procuramos linearidade ou proporcionalidade.
- **Utilidade (efetividade em decisões):** Métrica mais difícil: quantas *decisões acertadas* foram auxiliadas pela ferramenta. Podemos medir via estudo de caso (abaixo) se times que usaram ferramenta projetaram melhorias ou identificaram problemas que sem ferramenta não perceberiam. Ainda assim, podemos quantificar proxies: ex., número de insights gerados por sessão (perguntar ao usuário quantos "ahá!" teve), ou contagem de hipóteses de refatoração levantadas.

3) Estudos (Qualitativos e Controlados):

- **Estudo de Caso Real (Before/After):** Pegar um repositório de software real (preferência do usuário, ou open source) que passou por refatoração inspirada em algum conceito. Ex.: um sistema que era frágil, foi redesenhado focado em isolamentos (adotou bulkheads, circuit breakers). Rodar nossa ferramenta antes e depois, e ver se as analogias sugeridas "pre-refatoração" identificavam problemas (ex.: analogia com "organismo sem sistema imune" talvez), e pós refatoração se mapeia para "organismo com resposta imune robusta". Documentar se a ferramenta teria antecipado ou explicado a necessidade de mudança. Isso demonstra valor preditivo.
- **Experimento controlado com times:** Montar dois grupos de engenheiros para resolver um problema de arquitetura. Um grupo recebe nossa ferramenta com mapeamento para sistemas biológicos análogos, outro não. Verificar resultados: soluções do grupo com analogias são mais diversificadas? Pegaram aspectos que o outro não viu? Isso é difícil de quantificar isoladamente, mas podemos fazer avaliação qualitativa por jurados independentes das propostas de solução. Se conclusões do grupo "analogias" cobrem mais fatores (resiliência, ciclos feedback) e evitam certos erros que o grupo controle comete (p.ex. não ter fallback e resultando em cascata, que analogia teria prevenido lembrando de necrose), é evidência de utilidade.
- **Efeito 2ª ordem (Goodhart):** Também numa situação real, monitorar se desenvolvedores começam a "otimizar para o score" de forma tola – ex.: adicionar artifícios no código só para marcar caixas da analogia (seria análogo a metric gaming). Durante um período de adoção, coletar feedback anônimo: "você mudou algo apenas para melhorar score, sim/não?". Idealmente, a ferramenta deveria ser usada como diagnóstico, não KPI, mas precisamos vigiar.
- **Pesquisa de percepção:** Coletar de usuários (via questionário Likert) se as explicações e visualizações ajudaram a entender a analogia e se confiam no resultado. Por ex.: "Concordo que as analogias destacadas fazem sentido" e "A ferramenta me ajudou a ver meu sistema sob nova perspectiva".

4) Critérios de Falha (quando a analogia é perigosa ou enganosa):

Estabelecer condições para sinalizar ao usuário cuidado extra: - **Hot mas Inconsistente:** Se um par tem Hotness alto mas com uma dimensão muito baixa (ex.: S_{str} 0.9 mas S_{func} 0.2), isso sugere *falso alinhamento estrutural* [72] [3] . Critério: quando variância entre componentes > certo limite, marcar com ícone de alerta "Potentially Misleading Analogy – examine differences in roles/causality". - **Domínios muito distantes:** se domínios não compartilham princípios físicos ou lógicos, analogias podem ser puro acaso (ex.: analogia entre bolsa de valores e ecologia marinha – há trabalhos, mas enfim). Neste caso, sem ground truth pra validar, mas se a ferramenta sugerir analogia inusitada, encorajar validação: "Hipótese a ser testada – não aplicar diretamente". - **Checklist de Coerência (ver seção J item 7):** Implementaremos um checklist qualitativo: por exemplo, "Confirme se as premissas de causalidade realmente equivalem", "Verifique escalas: analogia retém validade se escala mudar?". Isso ajuda a evitar aplicação cega. - **Logs e revisão por pares:** Se ferramenta vai ser usada em consultoria ou ensino, sugerimos que analogias encontradas sejam discutidas em grupo – mitigando erro individual de interpretação.

Em suma, a validação combinará **benchmarks objetivos** (precisão, recall, estabilidade) e **avaliação contextual** (estudos com usuários). Estabeleceremos metas: por exemplo, *MVP exit criteria* – no OAEI Anatomy, atingir >=90% F1 comparado a aligners standard (para garantir que matching básico funciona). *V2 exit* – pelo menos 80% dos especialistas concordam com top 5 analogias sugeridas em casos testados. *V3 exit* – grupos com ferramenta apresentam 30% mais alternativas de design (indicando pensamento ampliado). Esses números podem ser ajustados, mas dão direções mensuráveis.

H) Visualização: Como Fazer Render "Top-Tier"

Desenharemos uma interface local-first que equilibra desempenho, clareza e exploração profunda. Componentes principais da UI:

- **Heatmap A×B:** O coração é uma matriz onde eixo X são conceitos do Domínio A e eixo Y do Domínio B (ou vice-versa). Cada célula representa a analogia entre conceito X_i e Y_j, codificada por cor = *hotness score*. Escolheremos um colormap perceptual uniforme (ex.: Viridis) para que intensidade corresponda fielmente ao valor [26] . Células "mais quentes" (alta analogia) serão destacadas talvez com contorno ou tamanho maior para chamar atenção. O heatmap será implementado em WebGL para suportar tamanho considerável e interações fluídas a 60fps mesmo com milhares de células [54] [55] .
- **Eixos etiquetados:** Os rótulos dos conceitos em A e B aparecem nas bordas. Precisaremos rotacionar ou scrollar se muitos. Podemos suportar *reordenar e agrupar* – ex.: agrupar conceitos de A por camada ou subdomínio. Isso ajuda visualmente a ver blocos (ex.: talvez todos conceitos de *imunidade* alinhem com todos de *segurança* e formem um submatriz quente).
- **Zoom e Pan:** Se matriz grande, usuário pode arrastar para focar numa região ou usar um mini-mapa thumbnail. Ao dar *zoom in*, revelaremos mais detalhes (via LOD: se muito distante, podemos esconder nomes e mostrar apenas blocos médios; ao aproximar, mostrar rótulos e valores). Com WebGL, podemos facilmente aplicar shaders para highlight de ranges.
- **Estabilidade visual:** Ao interagir (hover etc.), devemos manter a matriz estática exceto pelo highlight – nada de reordenar automaticamente ao filtrar (a não ser que usuário solicite), para não desorientar.

- **Drill-down por camada:** Um controle deslizante (slider ou botões Camada 0..N) permitirá filtrar quais camadas considerar no cálculo. Ex.: se usuário seleciona “Camada 0”, o heatmap recalcula considerando só atributos de camada 0 (conceitos base). Se “até Camada 2”, então hotness integra até dois níveis de subgrafo. Isso requer recalcular scores on-the-fly. Graças ao local compute, podemos fazer sob demanda para cada célula (talvez lento se muitos, então possivelmente pré-calculamos alguns snapshots ou calculamos progressivamente).

- Alternativamente, podemos ter heatmaps múltiplos: exibir lado a lado o heatmap para Camada0, Camada1, etc., talvez como pequenas matrizes ou superpondo isolinhas de “frio/quente” por camada. Mas isso complica. Mais simples: slider “incluir detalhes até nível k”. Ao aumentar k, espera-se que algumas correspondências esquentem (quando detalhes confirmam) ou esfriem (quando quebram).

- Destaque: se camada alta está selecionada e algumas células caírem de score significativamente vs camada básica, podemos colorir a borda de outra cor para indicar “esta analogia perde coerência nos detalhes”.

- **Painel Explain (“Explain & Align”):** Quando o usuário clica ou paira sobre uma célula (conceito A vs B), abriremos um painel lateral com a explicação detalhada:

- Listar as correspondências de subelementos: “**A** correspondendo a **B**” seguido de uma lista tipo:
    - *A1* — aligns with — *B1* (porque relação R de A e R’ de B correspondem)【some source】.
    - *A2* — no counterpart (isso evidenciaria uma quina: A tem elemento sem análogo).
    - *B3* — no counterpart on A side.
- Ou possivelmente exibir dois grafos lado a lado (subgrafos vizinhos de A e B) com linhas ligando nós correspondentes (como ferramentas de diff visual). Para não poluir, exibiremos subgrafo centrado nesses conceitos até certa profundidade (usuário poderia expandir mais se desejar).
- Mostrar os *scores por dimensão* (semântico, estrutural, etc.) possivelmente em forma de um gráfico de barras ou radar chart simples, para visual rápido do perfil.
- Texto narrativo: gerar frases salientando pontos chave: p.ex. “Ambos desempenham função de fail-safe isolando partes defeituosas: Apoptose remove célula danificada, Circuit Breaker isola serviço com erro 39 66 . Diferença: Apoptose envolve cascata molecular (caspases) sem claro equivalente no Circuit Breaker.” Fontes podem ser citadas aqui se disponível (ex.: do grafo evidence).
- **Cross-highlighting:** Ao passar mouse sobre um elemento no subgrafo de A no painel, destacar o correspondente em B (se houver) – pode ser mudar cor da borda do nó. Também simultaneamente, destacar essas posições no heatmap original: exemplo, se focamos Apoptose vs CircuitBreaker, e dentro dele Apoptose->Necrose (não mapeado), podemos piscar a célula Necrose vs (?) no heatmap para indicar “talvez analogia aqui” ou só para mostrar onde Necrose está na matriz (provavelmente alinhada com cascata de falhas ou nada – indicando lacuna). Isso conecta visão macro (heatmap) com micro (detalhe).

- **Interatividade extra:** usuário poderia ajustar mapping manualmente aqui – ex.: arrastar uma linha de correspondência errada para o nó correto. Isso seria feedback para recalcular score e refinar.

- **LOD/Tiling para grafos grandes:** Se A ou B tiverem muitos conceitos, a matriz pode ser 1000x1000 = 1,000,000 cells. Renderizar todos pontos individualmente é possível em WebGL, mas interação fica densa. Uma técnica: agrupar por clusters (k-means nos embeddings?) e mostrar cluster vs cluster de forma agregada quando zoomed out. Mas isso pode confundir sem interpretação clara. Uma

abordagem mais simples: se >N elementos, não mostrar todos nomes, apenas heatmass visual. Ao user dar zoom ou pesquisar nome, então mostrar região relevante.

- Alternativamente, se realmente muitos (como ontologia com 5000 classes vs outra), talvez melhor interface seja filtragem textual ou por propriedade antes. A ferramenta não visa exaustivamente todos conceitos de uma vez, mas explorando subset de interesse.
- Mas se precisarmos, usar *virtual scrolling* nas axes: ou seja, eixos roláveis, carregando os labels dinamicamente, mas heatmap calculado via shader sem precisar dom for each row/col.
- **Export e compartilhamento:** Permitir salvar imagens (PNG/SVG). Um cuidado: heatmap em canvas não fica em SVG facilmente; mas podemos converter. Talvez ofereça ambos: PNG para quick export, e um JSON do alinhamento para reuso (ex.: usuário pode salvar correspondências encontradas em arquivo para inserir num documento ou reutilizar).
- Dashboard local permitir exportar todo GraphSpec e mapeamentos descobertos – útil para writing papers ou difundir.

Desempenho: - Use WebGL (via frameworks como regl, deck.gl ou custom fragment shader) para matrix. Isso garante que mesmo milhares de cells são só um quad com shader iterando – muito rápido na GPU e leve na CPU. - Para highlights e interactions, precisamos pegar coordenadas do mouse e mapear à célula. Podemos ou renderizar cada célula com identificador codificado em cor no frame buffer invisível (picking technique) para obter ID instantaneamente. Ou calculamos posição pelo grid dimension (mais fácil se linear). - Manter 60fps: garantir que heavy computations (score recalculation) não acontece no thread principal travando UI. Possível usar web worker ou precompute e apenas ler. - **SVG vs Canvas vs WebGL:** - *SVG:* ótimo para crisp text e small number of shapes, degrade com milhares of objects. Vamos usar SVG para eixos se for umas poucas dezenas de labels; mas se centenas, possivelmente canvas text or WebGL text (which is trickier). - *Canvas:* fine for heatmap as an image, but static unless re-draw. We prefer GPU for dynamic update as slider moves. - *WebGL:* best for lots of objects (cells). For high DPI displays, ensures crisp lines if done right (but one must manage coordinate transforms carefully). - We can combine: e.g., use WebGL for main heatmap drawing, and overlay an HTML/SVG layer for tooltips and labels.

- **Avoiding chartjunk:**
- Keep visual minimalistic: no superfluous grid lines (maybe fine lines dividing cells if needed), consistent color scale with legend that has actual values labeled (0 = no analogy, 1 = perfect analogy).
- Provide context but not clutter: e.g., show domain names and possibly a summary of domain (like small description of system A and B on top).
- Use spacing and alignment to separate sections.
- **Accessibility:**
- Provide alternative text for heatmap summary (maybe auto-generate a sentence: "There are X high-analogy pairs, mostly between [group of concepts]").
- Color choices to be colorblind-friendly: Viridis is (tested for deuteranopia etc.) [27] . Also ensure contrast is sufficient (the darkest vs lightest are distinguishable even to monochrome).
- Possibly implement a switch to grayscale or pattern mode if needed.

- Text labels should be readable (watch font size, allow zoom UI scale).

- **UI Performance tuning:** Use requestAnimationFrame for updates, avoid heavy layout thrashing. Pre-compute as much as possible (like positions of cells).

- If very large matrices, consider splitting into tiles (256x256 for example) drawn as images to allow partial updates. If user pans, only update newly visible tile.
- Use binary texture upload to GPU if needed for dynamic values.

Design coherence: We want the UI to serve both quick insight and deep dive. So: - At a glance: user sees some bright spots → "hey, these correspondências são quentes". - Then can zoom in or click to get full explanation of particular spot. - We maintain linking between overview and detail always. - Use animation subtly: e.g., when clicking a cell, maybe gently highlight the row and column (like a cross) to emphasize which concepts are involved.

By following these practices and referencing known designs (like cluster heatmaps in bioinformatics tools), nossa visualização deve ser de “altíssimo nível” – não apenas bonita mas funcional e instintiva para o usuário técnico.

I) Pipeline Local + Input Variado: Importadores Plugáveis

Arquitetaremos a solução para rodar totalmente local, acomodar múltiplos tipos de input via módulos plugáveis, e atualizar incrementamente com mínimo esforço do usuário. Componentes principais do pipeline:

- **Importers (Importadores):** Conjuntos de componentes que pegam um formato de input e produzem GraphSpec canônico. Teremos:
- `import_text` – lê arquivos de texto (Markdown, PDF? Pode ser) e aplica NLP para extrair conceitos e relações. Por simplicidade MVP, podemos suportar texto estruturado via marcações especiais ou formatos limitados (ex.: YAML list of concepts + relations), mas a ideia é evoluir para NER + relation extraction.
- `import_repo(language)` – analisa repositório de código. Para Python (prioritário), usa AST do Python: encontra classes, funções, decoradores (ex.: @circuit_breaker -> tag?), dependências (import statements), e interações (função chama outra -> edge "calls"). Também extrai docstrings talvez para conceituar propósito das funções. Tudo isso vira grafo: e.g., cada classe ou módulo = node com edges is_a (herança), part_of (estrutura de pasta), calls (fluxo).
- `import_ontology` – carrega ontologia (OWL, RDF, TTL) usando alguma lib (rdflib). Cria nós para classes e propriedades, edges for subclass, part-of, etc. Podem ser grandes, então talvez filtrar por relevância (ex.: user indica um top concept e importamos só subárvore dele).
- Possível outros: `import_json` para um JSON já parecido com GraphSpec (facilita integração de qualquer fonte arbitrária, se user prepara).

- Importers serão **plugins** no sentido de fácil extensão: define interface (input -> GraphSpec). Podemos permitir que usuários escrevam novos importers (e.g., for Java using JavaParser, or for other domain knowledge).

- **Armazenamento Local:**

- Usaremos um banco leve tipo SQLite para persistir grafo e possivelmente correspondências. Alternativamente, como GraphSpec é JSON, poderíamos armazenar JSON files. Mas SQLite nos dá consultas (ex.: procurar conceito por nome, join arestas).
- Modo de operação: quando usuário roda import, salvamos GraphSpec no banco (ou file). Se rodar de novo, atualizamos (ou mantém versões).
- Para dados binários (ex. embeddings model?), podemos ter diretório `.analogy_tool/` com arquivos (um modelo .bin por domínio).

- Tudo local, sem envio a servidor. Isso garante privacidade: repositórios privados seguros.

- **Backend de cálculo (FastAPI ou similar):**

- Uma pequena aplicação local (pode ser CLI que abre um servidor REST ou WebSocket) que faz as operações intensivas (calcular hotness, run matching, etc.) sob demanda da UI.
- Por exemplo, UI (frontend in electron or browser) faz GET `/align?A=BioGraph& B=SoftGraph` e backend retorna JSON de alignment scores, que UI então renderiza.
- FastAPI seria bom, ou Flask. Ou mesmo tudo dentro Electron main process. Mas separar preocupa com concurrency ou heavy computing no UI thread, então um backend separate is wise.

- Esse backend pode também agendar re-import tasks (like watcher: "if new commit, re-run import_repo").

- **Incremental Jobs:**

- Quando uma importação é acionada, marcar timestamp. O alignment job pode então ver: se Graph A or B mudou (new timestamp), recomputar; senão, usar cache.
- Para importers de repositório, podemos integrar com git hooks ou um cron local: e.g., "Modo CI local": a ferramenta poderia ser configurada para rodar toda semana, analisa as mudanças do repo e recalcula trends (ex.: hotness do sistema com analogia X subiu ou desceu).
- O pipeline pode permitir comparações de versão: ex.: user selects two versions of same graph to see differences em analogias.

- Ao importar texto, possivelmente permitimos merging: se user tem vários documentos (cada um GraphSpec), podemos uni-los num só grafo (preferir, para central).

- **Local UI e orchestration:**

- Podemos ter uma CLI (command-line) para importar e output some summary: e.g. run `analogy import -t biology.txt -d Biologia` e `analogy import -r ./mySoftware` e depois `analogy analyze Biologia mySoftware`.
- A UI gráfica (dashboard) pode chamar esses via APIs.

- Para persistência de settings e caches, use arquivo config local (json or yaml) e DB.

- **Plugins de importação:**

- Estrutura de plugin: talvez usar Python entry points or a plugin manager. For now, since it's local, the user can drop a `.py` file implementing `def import(data) -> GraphSpec` and register in config. e.g., user in finance domain can add importer for “Excel spreadsheets of risk model” etc.
- We define a base class or interface. Provide examples with our built-ins.

Lidando com Input Variado sem perder coerência: - Diferentes fontes podem have overlapping concept names but different granularity. Precisamos normalizar e merge. Faremos uma fase pós-import chamada **normalization & merge**: que unifica nós duplicados and merges edges. Perhaps prompt user if uncertain: “Concept X from text seems same as Class Y from code, merge?”. - Coerência do grafo: - Use consistent naming: e.g., ensure every node has domain prefix to avoid collision (we did "BIO_" "SW_"). - Use same ontology of relation types across importers (we define that centrally). - If some importers give contradictory info (one says A part_of B, other says A is different), maybe just include all and let analogies highlight conflict (or unify if one clearly wrong). - We might keep separate subgraphs for separate sources initially to avoid contamination, then allow edges to be added between if match found (like linking ontology class to code class if names align).

CI Local Mode: - The idea: As code evolves, you'd like to see trends: e.g., *Analogical Hotness to some ideal might increase.* We can log metrics each run (store in SQLite timeseries). - Possibly provide a command `analogy trend domainA domainB conceptX` to get how hotness of conceptX to something changed over time. Or output a sparkline on UI. - This helps detect when a design change improved alignment with desired analogy (maybe a target analog system known to be robust). - Also helps ensure metric does not degrade unexpectedly (maybe highlighting regressions).

Security & Privacy: - No data leaves machine, user controls input. - If using FastAPI, ensure it binds to localhost only, and maybe require token for API calls to avoid other local processes messing (but since it's local user, not huge risk). - If we include any pre-trained model, bundle it or download with user consent, not call external API.

Performance on import: - AST parsing of code could be heavy for large repos, but we can incremental parse: only changed files (if we store last parse). - NLP on big documents uses CPU, maybe do basic stuff or let user provide curated input for important parts to reduce noise. - Possibly allow user to mark concept anchors in code (via comments) to improve import accuracy.

In design, pipeline tries to be robust: user can always manually edit GraphSpec JSON if something wrong, and reload it. That’s advantage local: all data visible and editable by user if needed.

To illustrate pipeline:

```
User Input (Texto, Código, Ontologia)
   -> [Importer correspondente]
         -> GraphSpec (JSON)
            -> [Normalize/Merge] -> Master Graph(s) stored
   -> [Aligner] (with optional cached results)
         -> Alignment results (scores, matches)
   -> [Visualization] (heatmap, etc.)
```

All steps under user's control, with intermediate output accessible. Isso reforça confiabilidade (user pode inspecionar e ajustar), e offline by design.

J) Entregáveis do Projeto

Nesta seção compilamos artefatos e planos concretos que resultam do trabalho, úteis para implementação e acompanhamento.

1) Mapa de Lacunas (GAP MAP) – Prioridades e Ações: *(Com base na seção A, resumimos principais lacunas em formato acionável.)*

Lacuna/Questão	Impacto	Urgência	Ação Proposta	Status/Notas
Definição formal de analogia "quente" e critérios de coerência	Alto	Imediata (antes de dev)	Revisar literatura (Gentner SMT, Holyoak) e definir métrica vetorial. Refinar com feedback de especialista.	**Em Progresso** – Glossário B definido, precisa validação.
Modelo de grafo multi-camada unificado (GraphSpec)	Alto	Imediata (base do dev)	Especificar JSON GraphSpec e implementar import básico (texto simples, AST Python).	**Concluído (draft)** – Esquema JSON proposto na seção D; implementação inicial em andamento.
Algoritmo de matching estrutural (greedy)	Alto	MVP	Codar matching guloso + heurística vizinhança. Testar em ontologia conhecida.	**Pendente** – Planejado para Sprint 1.
Visual heatmap performático e intuitivo	Alto	MVP	Implementar protótipo WebGL heatmap, usar dataset sintético para teste de desempenho e paleta.	**Em Progresso** – Experimento WebGL com 500x500 ok; integrando colormap Viridis.
Validação com casos reais	Alto	Paralelo ao dev	Montar dataset de ~20 analogias (10 boas, 10 ruins) com avaliação especialista. Agendar revisão após MVP.	**Em planeamento** – Identificando especialistas dispostos.
Goodhart's Law – evitar uso indevido do score	Médio	v2	Adicionar avisos na UI, remover qualquer linguagem de "metas/ OKR" associada ao score.	**Pendente** – A incluir no design UX.

Lacuna/Questão	Impacto	Urgência	Ação Proposta	Status/Notas
Plugin import código para JS/Java	Médio	v2/v3	Extender importer para suportar mais linguagens (usar JSIde parser ou srcML).	**Backlog** – Foco inicial Python; outros via demanda.
Aprendizado de pesos (ML)	Médio	v3	Coletar dados de alinhamentos confirmados, treinar modelo de similaridade. Opcional se heurísticas forem suficientes.	**Backlog** – Avaliar após suficientes casos coletados.
UI Drill-down por camada + explicações detalhadas	Médio	MVP/v2	MVP: Mostrar breakdown por dimensão; v2: adicionar slider de camada e subgrafo comparativo no painel.	**MVP Básico em dev** – Painel explicação textual simples. Slider previsto v2.
Monitoramento de tendências (CI local)	Baixo	v3	Implementar registro histórico de hotness e pequeno módulo de comparação de versões.	**Backlog** – Idea registrada, fazer se tempo permitir.

(O mapa acima orienta a equipe nas próximas sprints, priorizando o essencial para MVP e planejando evoluções.)

2) Modelo Canônico (GraphSpec) + Exemplos Mínimos:

Conforme seção D, formalizamos o GraphSpec JSON para representar conceitos em camadas, com nós, arestas e metadados padronizados. Um exemplo mínimo de dois grafos (Biologia vs Software) foi fornecido [39] [66].

Resumo: - **Nodes:** `id, label, domain, layer, parents, tags, embedding` – ex.: `{"id":"BIO_APOPTOSIS","label":"Apoptosis","layer":0,"tags":["FailureMode:FailSafe","VSM:S1"]}`. - **Edges:** `source, target, type, weight, evidence` – ex.: `{"source":"BIO_CELL_DAMAGE","target":"BIO_APOPTOSIS","type":"triggers"}`. - **Graph Meta:** `origin, domain, version, ...` – ex.: `{"origin":"myRepo v1","domain":"Software"}`.

Esse modelo serve como contrato entre importadores e algoritmos. Já implementamos um *draft* do schema e um exemplo concreto, que servirá de teste unitário. Com GraphSpec, podemos facilmente serializar/deserializar em JSON (portanto, integrar com JS frontend ou outras linguagens).

3) Hotness Score – Fórmula + Explicação + Por Camada:

Nosso *hotness* é calculado combinando quatro dimensões:

$$H_{\text{comp}} = w_{\text{sem}} S_{\text{sem}} + w_{\text{str}} S_{\text{str}} + w_{\text{func}} S_{\text{func}} + w_{\text{scale}} S_{\text{scale}},$$

onde: - $S_{\text{sem}} =$ similaridade semântica (embedding coseno ou lexical) [1] , - $S_{\text{str}} =$ correspondência estrutural (relações mapeadas / total) [48] , - $S_{\text{func}} =$ alinhamento de papéis funcionais (tags VSM e posição em loops) [73] [8] , - $S_{\text{scale}} =$ compatibilidade de escala temporal e tipo de falha.

Pesos iniciais $w_i = 1$ (normalização linear). Exemplo de interpretação: se analogia tem $S_{\text{sem}} =0.2, S_{\text{str}}=0.9, S_{\text{func}}=0.8, S_{\text{scale}}=0.7$, então

$$H_{\text{comp}} = 0.2 + 0.9 + 0.8 + 0.7 = 2.6 \,(\text{de } 4.0) = 0.65 \,(\text{normalizado}).$$

Explicação gerada: "Score 0.65 – Relacional e funcional fortes, semântica fraca (nomes diferentes), boa correspondência em escala." Cada componente será exibido ao usuário, possivelmente com coloração diferente no painel, facilitando entender a composição.

Hotness por camada: definimos H_d = score considerando até camada d. Implementação: ao calcular S_{str} e S_{func}, limitamos a vizinhança até d níveis de parentes/filhos. Assim vemos a progressão. Exibiremos na UI um controle para d, e possivelmente um gráfico. Se H_0 ≈ 1.0 mas H_2 despenca, usuário percebe analogia superficial. Isso integra no processo decisório (não confiar apenas no número global).

Evitar falsos positivos: adotamos regra: se uma dimensão contradiz fortemente (ex.: $S_{\text{func}} < 0.2$ enquanto outras >0.8), marca analogia como potencialmente inconsistente [72] . No código, podemos por threshold e flag.

4) Plano de Validação (Provar que Funciona):

- **Conjunto de teste**: 20 analogias cross-domain rotuladas (ex.: 10 boas, 10 ruins) – já sendo compilado via especialistas.
- **Teste unitário**: rodar ferramenta nesses pares, verificar se as boas têm hotness > threshold T e ruins < T (escolher T ~0.5 para start). Ajustar se necessário.
- **Comparativo com baseline**: em ontologia alignment (Anatomy dataset), rodar nosso matcher vs. AML/LogMap; esperar similar precision/recall (>0.9). Documentar resultados.
- **User study**: após MVP, selecionar 5 devs, 5 cientistas; dar a eles cenários e a ferramenta. Coletar: se analogias fizeram sentido (questionário), se levaram a insights novos. Também monitorar uso (logs locais com consentimento) para ver quais features clicaram (ex.: drill-down usage, indicates value).
- **Simulação de 2ª ordem**: rodar pipeline regularmente em um projeto durante desenvolvimento (por ex., monitorar um repositório open-source ativamente desenvolvendo resiliencia). Ver se metric trending correlaciona com release notes (ex.: "refatoramos para ser mais modular" – veremos algum score modularidade subir?).
- **Gate para release**: somente lançar v1 publicamente se: (a) Nenhum falso positivo crítico nos testes (ex.: sugerir analogia absurda com score alto sem alerta), (b) Usuários de teste expressam confiança básica nos top resultados, (c) Performance: interface lida com tamanho alvo (digamos 200x200 matrix) sem travar.

5) Roadmap MVP → v2 → v3:

- **MVP (v1.0)** – *Foco:* funcionalidade básica e valor imediato.

- **Importadores:** suporte a pelo menos 2 fontes (ex.: texto estruturado e Python AST).
- **Matching:** algoritmo guloso + similares semânticos; output de alinhamento básico.
- **UI:** heatmap interativo com highlight e valores, painel explicativo simples textual, sem drill profundo.
- **Validação:** teste em 2-3 casos reais e review com stakeholders.
- **Critério de sucesso:** Usuário consegue pegar seu sistema e um análogo e ver correspondências não triviais emergindo. MVP previsto em ~3 meses.
- **v2.0** – *Foco:* refino e robustez.
- **Matching:** introduzir Hungarian/ILP para evitar erros de assignment; penalidades de incoerência.
- **Dimensões extras:** incorporar timescale/failure explicitamente (talvez user annota).
- **UI:** slider de camada, grafo comparativo no painel, opções de agrupar/filtrar conceitos. Colormap e acessibilidade finalizados.
- **Plugins:** import de ontologia, possivelmente import básico para outra linguagem (JS).
- **Validação:** release beta para grupo maior, coletar métricas de precisão/recall e feedback UI.
- **Docs & Extensibilidade:** escrever documentação para plugin import e formato GraphSpec para que outros contribuam.
- **Timeline:** +3-4 meses após MVP.
- **v3.0** – *Foco:* inteligência adaptativa e integrações.
- **Aprendizado:** treinar modelo de similaridade ou ajustar pesos via feedback (interações do usuário ou dataset).
- **Predição proativa:** sugerir analogias interessantes automaticamente (ex.: “Seu sistema parece muito com X biológico, veja...” sem que usuário tenha fornecido X – buscar em biblioteca interna de grafos).
- **Collab & Versão:** permitir comparar 2 sistemas lado a lado (ex.: seu sistema vs sistema ideal de referência, ou antes vs depois).
- **CI mode:** background job monitoring repository, output summary (maybe commit messages with "hotness changed +0.1 on modularity analogies").
- **UX polimento:** relatórios exportáveis, talvez modo apresentação (para usar em workshop, com snapshots de grafo).
- **Timeline:** +6 meses.
- **vX (além):**
- Possível v4: banco de grafos análogos (library de sistemas biológicos e de software) para sugerir analogias cross-empresa.
- v5: ferramenta comunitária, compartilhamento de grafos (ainda local-first mas com sync opt-in).

Cada versão terá critérios de aceite medidos (ex.: v2 deve aumentar recall em X%, v3 modelo ML deve melhorar ranking em testes cegos etc.).

6) Lista de Leitura Essencial (10–30 fontes):

Para fundamentar e aprofundar o trabalho, as seguintes fontes são cruciais, cada qual contribuindo para aspectos teóricos ou práticos:

1. **Gentner (1983), *Structure-Mapping: A Theoretical Framework for Analogy*** – Artigo seminal definindo analogia como mapeamento relacional, introduzindo princípio da sistematicidade [1]. Fundamenta nossa abordagem estrutural.

2. **Falkenhainer et al. (1989), *The Structure-Mapping Engine*** – Detalha o SME, algoritmo computacional para analogias baseadas em Gentner. Fornece insights sobre implementação e limitações (necessidade de representações manuais) que buscamos superar.
3. **Turney (2008), *The Latent Relation Mapping Engine*** [4] [5] – Mostra como combinar aprendizado estatístico (LRA) com analogia: descobre analogias via corpora. Inspira uso de embeddings e dá baseline de performance humana vs modelo em analogias.
4. **Shvaiko & Euzenat (2013), *Ontology Matching*** – Livro abrangente sobre técnicas de alinhamento de ontologias. Útil para nossos algoritmos de correspondência e avaliação (precision/recall, OAEI benchmarks).
5. **Pesquita et al. (2014), *Towards Visualizing the Alignment of Large Biomedical Ontologies*** – Explora desafios de visualizar correspondências em ontologias grandes [48] [17]. Informa nosso design de interface (heatmaps, highlight interativo).
6. **Ashby (1956), *An Introduction to Cybernetics*** – Introduz Lei da Variedade Requisita [6] e conceitos base de controle e complexidade. Fornece base teórica para incluirmos variedade e feedback.
7. **Beer (1972), *Brain of the Firm* (VSM)** – Define o Viable System Model. Embora denso, é a base de nossos tags S1–S5 e entender organização recursiva. Complementado por artigos modernos (e.g. Gorelkin 2025 [73] [8]).
8. **Nygard (2007), *Release It!*** – Livro que popularizou padrões de robustez em software (Circuit Breaker [67], Bulkhead, etc.). Nos dá contexto e motivação prática para analogias (muitos desses padrões têm contrapartes biológicas).
9. **Fortuna et al. (2011), *Evolution of a modular software network*** – Estudo PNAS comparando evolução do Debian a ecossistemas [16] [17]. Prova empírica de analogias bio-software (modularidade vs robustez). Dá confiança de que nosso projeto tem fundamento.
10. **Holyoak & Thagard (1995), *Mental Leaps: Analogy in Creative Thought*** – Livro cobrindo analogia sob ótica cognitiva e criativa. Introduz ACME (mapeamento por restrições) e discute usos e abusos de analogias em ciência. Ajuda a moldar nossas preocupações com analogias enganosas.
11. **Hummel & Holyoak (2005), *Distributed representations of structure: LISA*** – Explica um modelo conexionista de analogia, lidando com ligação de variáveis via sincronização. Oferece perspectiva alternativa e possivelmente ideias para ML approach na v3.
12. **Valerie Aurora (2019), *Scaling Python services with control theory* (QCon talk)** – Exemplo prático aplicando controle em sistemas (PID loops para autoscaling) [69] [65]. Reforça relevância de controle teórico em dev ops, reforçando interseção de domínios do projeto.
13. **Dedre Gentner & Colleagues (varios), *Analogy and Metaphor* (Stanford Encyclopedia of Philosophy, 2022)** [1] [2] – Síntese atualizada das teorias de analogia, incluindo críticas (quando sistematicidade falha [21]). Ajuda a balizar nosso modelo teórico e evitar overreliance em um só critério.
14. **Tufte (1983), *The Visual Display of Quantitative Information*** – Clássico de visualização. Lembra-nos de maximizar razão sinal/ruído, evitar *chartjunk*, escolher boas escalas. Aplica-se no design do heatmap e explicações claras.
15. **Cabo & others (2020), *The misuse of colour in science communication* (Nature Comms)** [26] [28] – Mostra como colormap inadequado distorce dados e exclui público daltônico. Fundamenta nossa escolha de paleta e serve para justificar tecnicamente ao usuário (caso perguntem "posso ter rainbow?" – responderemos com essa referência).
16. **Goodhart (1975), *Problems of Monetary Management*** – Origem do Goodhart's Law. Importante para justificar nossa cautela em metricização do hotness. Junto com Marilyn Strathern (1997) que reformulou: "quando medida vira alvo, deixa de ser boa" – alerta aplicado no projeto para não virar KPI cego.

17. **Euzenat et al. (2022), *BERTMap: A BERT-based Ontology Alignment System*** [47] [15] – Exemplo de uso de transformers para alinhamento. Orienta se quisermos incorporar BERT embeddings para similaridade semântica de rótulos complexos (especialmente em ontologias biomédicas).
18. **Gelernter (1998), *Machine Beauty: Elegance and the Heart of Technology*** – Livro filosófico que toca em analogias em design de software. Pode inspirar narrativa e enfatiza porque analogias importam para criatividade e elegância.
19. **Resnick (2017), *Advice from a Caterpillar (on emergence)* – (Hipotético)**. [Colocar aqui referência sobre emergência e analogias se achar, ou sobre degeneração funcional].
20. *(Espaço reservado caso precisemos incluir alguma referência interna de documentação ou regulamentação, ou relatórios de teste).*

(As referências acima cobrem teoria de analogia, métodos de alinhamento, visualização e estudos de caso – uma base sólida para implementar e validar o projeto.)

7) Checklist de Coerência (Para evitar analogias enganosas):

Antes de tirar conclusões de uma analogia identificada, use esta lista de verificação:

- [ ] **Mapeamento relacional completo?** – Verifique se a maioria das relações importantes do domínio-fonte encontram correspondência no domínio-alvo. Se muitas ficam de fora, a analogia pode ser superficial.
- [ ] **Consistência Causal/Funcional?** – As causas e efeitos correspondentes realmente equivalem? (Ex.: "X causa Y" em A e "X' causa Y"' em B – a natureza dessas causalidades é análoga ou meramente coincidência?). Se divergirem (positivo vs negativo, linear vs não-linear), cuidado.
- [ ] **Alinhamento de Papéis** – Os elementos mapeados ocupam funções similares nos sistemas? (Ex.: ambos são mecanismos de defesa, ou um é defesa e outro é ataque? Se for o segundo, analogia invertida poderia levar a erro).
- [ ] **Escala Compatível** – As dinâmicas se dão em escalas comparáveis? Se um sistema opera microsegundos e outro em anos, considerar impactos de escala nas conclusões – talvez precise escalonar analogia ou abstrair nível.
- [ ] **Diferenças explícitas identificadas** – Confira a seção "onde quebra" da explicação. Cada analogia terá pontos não mapeados; entenda-os. Pergunte: essas diferenças invalidam a transferência de ideia? Ou são detalhes implementacionais?
- [ ] **Não confundir metáfora motivacional com isomorfismo rigoroso** – Uma analogia pode ser útil apenas como metáfora para pensar, mas não sugerir implementações diretas. Se for esse o caso (ex.: "organização como um jardim" – bonito, mas não equacionável), use apenas como inspiração, não blueprint.
- [ ] **Validação Empírica** – Para qualquer insight de design obtido, tente validá-lo com um pequeno experimento ou protótipo. Goodhart: não mude todo sistema só para aumentar hotness sem evidência real de melhoria.
- [ ] **Consulta Cruzada** – Se possível, converse com um especialista do outro domínio. Ex.: se encontrou analogia com um processo biológico, cheque com um biólogo se a interpretação está correta. Muitas analogias falham por simplificações erradas do domínio de origem.
- [ ] **Atualização de Modelo** – Se analogia parece errada, registre feedback na ferramenta (desmarcar correspondência, ajustar tag). Assim, evita voltar a aparecer e melhora futuras sugestões (no v3 com aprendizado).

- [ ] **Contextualize** – Considere o contexto de cada sistema: restrições de engenharia vs de evolução biológica diferem. O que é ótimo na natureza pode ser impraticável no software por custo ou timing, e vice-versa (natureza não tem engenheiro central, software tem).
- [ ] **Não forçar analogia** – Se precisar torcer muito fatos para caber na analogia, provavelmente não é útil. Lembre: analogias servem a você, não você a elas. Abandone analogia quando começar a confundir mais do que esclarecer.

Usando este checklist, esperamos evitar derivações indevidas e manter as analogias "nos trilhos". Ele será incorporado na documentação do usuário e possivelmente em prompts dentro da UI (ex.: ao atingir hotness muito alto, apareça lembrete: "Verifique critérios de coerência antes de prosseguir").

K) Limites e Honestidade

É importante reconhecer onde nosso modelo e abordagem carecem de consenso ou garantias, e como lidamos com essas incertezas:

- **Teoria de analogia não unificada:** Embora adotemos SMT de Gentner, há visões divergentes (ex.: abordagem de *holonomia* de Hofstadter, ou ideia de que analogia envolve mais pragmática). Não há consenso absoluto sobre como humanos avaliam analogias [74]. Nossa escolha estrutural pode falhar em casos onde contexto ou objetivos do usuário importam mais. *Mitigação:* manter flexibilidade – permitimos pesos ajustáveis e incorporamos *pragmatic constraints* (ex.: relevância para meta do usuário) em futuras versões, inspirado por trabalhos como Holyoak que fala de *pragmatic alignment* [75].
- **Medir "hotness" objetivamente:** Não há na literatura uma métrica padrão de analogia de 0 a 1. Criamos uma. Pode ser criticável: *"por que 4 dimensões e não 6?"* ou *"pesos deveriam ser exponenciais"*. *Honestidade:* admitimos que a fórmula é heurística inicial – calibrada empiricamente, não derivada de teoria fechada. Planejamos refiná-la via experimentos (feedback supervisionado, sec. F v3). Enquanto isso, apresentaremos *hotness* como *"score experimental"*, não verdade divina.
- **Complexidade computacional potencial:** Graph matching pode ser NP-difícil no caso geral. Em pior caso, comparar grafos grandes com muitas permutações seria intratável. Nós contornamos restringindo problema (focos locais, top-K candidatos) – mas isso significa que poderemos perder alguma correspondência global ótima. É trade-off: priorizamos rodar local, então assumimos grafos não gigantes e estrutura modular (o que condiz com nossos domínios de interesse, tipicamente). *Honestidade com usuário:* se ele tentar comparar dois grafos enormes e nota demora ou falta de resultado, explicaremos limite e sugeriremos restringir escopo ou resumir input.
- **Falta de ground truth extenso:** Diferente de tradução ou reconhecimento de imagem, não há dataset enorme de analogias certo-errado para treinar/verificar. Nossas validações serão limitadas a casos montados. Isso significa que certos analogias sugeridas podem ficar sem verificação externa. *Mitigação:* encorajar usuários a validar em seu contexto (checklist acima). E manter no software claro que ele sugere, não garante.
- **Bias nos embeddings e dados:** Se usamos embeddings pré-treinados (ex.: word2vec), eles carregam vieses (ex.: "executivo" perto de "homem" e "enfermeira" perto de "mulher", etc.). Isso poderia afetar analogias sugeridas (talvez mapeando coisas de forma estereotipada). *Mitigação:* usar embeddings técnicos/domínio quando possível e permitir ajustar. Exibir evidências concretas das correspondências para usuário avaliar, ao invés de "modelo disse".
- **Analogia vs Homologia literal:** As vezes a ferramenta pode achar analogia quando na verdade é quase identidade (literal similarity [70]). Ex.: se importarmos duas versões diferentes da mesma

ontologia, vai marcar tudo 1.0 – trivial e não útil. Precisamos detectar e talvez filtrar casos triviais (por nome idêntico e contexto idêntico) para focar no novo. *Comunicação:* se usuário usa ferramenta em dois sistemas muito próximos (ex.: microservice A vs B quase clonados), dizer que analogia alta era esperada, não confundir com insight.

- **Segunda Ordem (efeito do uso):** Já coberto mas reforçando: se equipe otimiza design só para melhorar score, pode introduzir complexidade desnecessária ou atrasar entregas (Goodhart). Reconhecemos que *qualquer métrica de arquitetura é sujeita a isso*. Por isso, enfatizamos no manual: *"Não use o hotness como meta absoluta; use-o para refletir e explorar"*. E iremos monitorar (com autorização) usos para detectar sinais de desvio (ex.: commit messages "increase analogical score" seriam alarmantes).
- **Limites de visualização:** Por melhor que seja o heatmap, visualizar mais que algumas centenas de itens fica desafiante cognitivamente. O usuário pode não perceber uma correspondência importante se estiver perdida num mar de células. Não há solução mágica – resumir demais perde detalhe; detalhar demais perde big picture. Vamos iterar no design, talvez introduzir funcionalidades como "mostrar top 10 analogias de cada lado" para garantir que nada crucial passe despercebido. Mas ainda assim, humanos têm limites de atenção. *Honestidade:* nossa ferramenta não substitui análise humana focada; é um auxiliar para revelar padrões, mas requer curadoria do usuário para extrair história completa.
- **Aplicabilidade geral:** Estamos assumindo que comparar software com biologia (e org) é útil. Há risco de algumas analogias serem forçadas ou irrelevantes em certos contextos de software (nem todo software quer ser "vivo"). Se assim, a ferramenta poderia ser vista como curiosidade sem utilidade prática. Precisamos provar utilidade via casos reais (validação 3). Se descobrirmos que analogias não ajudaram decisões, seremos transparentes e ajustaremos foco ou funcionalidades. Talvez a ferramenta se torne mais um *educational toy* se for o caso – mas nosso objetivo é produto útil.
- **Consenso inexistente em alguns paralelos:** Ex.: VSM não é universalmente aceito como modelo perfeito (tem críticas, p. ex. de ser difícil mapear na prática). Ao usá-lo para tags, podemos encontrar objeções de outras escolas de pensamento organizacional. Abordagem: usar VSM como uma das perspectivas, mas não a única. Permitir tags custom (talvez alguém queira taggear componentes com arquetipos diferentes – ex.: "modelo Cynefin" ou "CNC metade"). Deixar espaço para extensibilidade e deixar claro que escolhemos VSM por conveniência teórica, não porque seja a *verdade* absoluta da cibernética.
- **Incerteza quantitativa nos scores:** Dar um número pode iludir precisão. Vamos comunicar incerteza: talvez exibir ranges ou simplesmente adotar 1 casa decimal e evitar falsa sensação de exatidão (ex.: dizer 0.8 em vez de 0.832). Poderíamos até calcular uma incerteza se variarmos pesos dentro de margem e ver variação – ou se a entrada tiver ruído.

Em suma, manteremos postura de *"confiança humilde"*: acreditamos que o sistema revelará insights, mas nós mesmos questionamos e testamos suas saídas. Documentaremos as limitações no manual do usuário. Faremos questão de no onboarding do produto explicar que analogias são ferramentas de pensamento, não soluções automáticas.

Com essas considerações, seguimos construindo o projeto de forma rigorosa e transparente, prontos para ajustar rota conforme evidências e feedback – assim, unindo a busca científica de entendimento com a engenharia prática de produto útil, sem prometer milagres mas entregando inovação sólida. 1 16

---

1 2 3 21 22 72 74 75 Analogy and Analogical Reasoning (Stanford Encyclopedia of Philosophy/ Summer 2022 Edition)
https://plato.stanford.edu/archives/sum2022/entries/reasoning-analogy/

4 5 45 46 [0812.4446] The Latent Relation Mapping Engine: Algorithm and Experiments
https://arxiv.org/abs/0812.4446

6 9 Requisite Variety - an overview | ScienceDirect Topics
https://www.sciencedirect.com/topics/computer-science/requisite-variety

7 8 11 12 37 38 57 58 59 60 73 Stafford Beer’s Viable System Model for Building Cost-Effective Enterprise Agentic Systems | by Mikhail Gorelkin | Nov, 2025 | Medium
https://medium.com/@magorelkin/stafford-beers-viable-system-model-for-building-enterprise-agentic-systems-81982d6f59c0

10 18 Aligning ontologies describing computer science for patents and scientific papers
https://repositum.tuwien.at/bitstream/20.500.12708/168404/1/Marcher%20Hannes%20-%202023%20-%20Aligning%20ontologies%20describing%20computer%20science%20for...pdf

13 Graph neural network based on graph kernel: A survey - ScienceDirect
https://www.sciencedirect.com/science/article/abs/pii/S0031320324010586

14 15 47 arxiv.org
https://arxiv.org/pdf/2112.02682

16 17 19 20 48 Evolution of a modular software network - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC3250116/

23 24 25 54 55 shinyheatmap – Ultra fast low memory heatmap web interface for big data genomics | RNA-Seq Blog
https://www.rna-seqblog.com/shinyheatmap-ultra-fast-low-memory-heatmap-web-interface-for-big-data-genomics/

26 27 28 56 The misuse of colour in science communication | Nature Communications
https://www.nature.com/articles/s41467-020-19160-7?error=cookies_not_supported&code=3b71ad4d-615e-4c6d-a14e-4a74337899e0

29 Goodhart's law - Wikipedia
https://en.wikipedia.org/wiki/Goodhart%27s_law

30 31 32 36 70 71 Structure-mapping theory - Wikipedia
https://en.wikipedia.org/wiki/Structure-mapping_theory

33 34 35 groups.psych.northwestern.edu
https://groups.psych.northwestern.edu/gentner/papers/Gentner83.2b.pdf

39 40 Apoptosis - Wikipedia
https://en.wikipedia.org/wiki/Apoptosis

41 42 43 44 Control loop - Wikipedia
https://en.wikipedia.org/wiki/Control_loop

49 50 51 Limitations of Graph Neural Networks | by Sergei Ivanov | TDS Archive | Medium
https://medium.com/data-science/limitations-of-graph-neural-networks-2412fffe677

52 [PDF] Subgraph Isomorphism - CSE - IIT Kanpur
https://www.cse.iitk.ac.in/users/dsrkg/cs210old/cs245/html/seminar/isomorph.pdf

53 [PDF] Graph Alignment Kernels using Weisfeiler and Leman Hierarchies
https://proceedings.mlr.press/v206/nikolentzos23b/nikolentzos23b.pdf

61 [PDF] Autoscaling cloud resources with real-time metrics
https://journalwjarr.com/sites/default/files/fulltext_pdf/WJARR-2025-1660.pdf

62 Investigation into Auto-scaling Mechanisms in Cloud Computing
https://link.springer.com/10.1007/978-981-95-3061-8_21

63 Auto-scaling Approaches for Cloud-native Applications - arXiv
https://arxiv.org/html/2507.17128

64 65 69 PID Loops and the Art of Keeping Systems Stable - InfoQ
https://www.infoq.com/presentations/pid-loops/

66 67 Circuit Breaker
https://martinfowler.com/bliki/CircuitBreaker.html

68 Bulkhead pattern - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead