



Ferramenta Local-First de Mapa de Calor de Analogias – Especificação Completa

A) PRD – Documento de Requisitos do Produto

Problema: Profissionais e pesquisadores muitas vezes desejam **comparar conceitos complexos de domínios distintos** (ex.: biologia vs. arquitetura de software) para encontrar **analogias úteis**, porém não dispõem de ferramentas sistemáticas para isso. Hoje, a busca de analogias é manual e sujeita a vieses, tornando difícil identificar padrões entre sistemas vivos, sistemas de software e organizações. O problema agrava-se com informações dispersas em diversos formatos (textos, ontologias, código), dificultando construir uma **visão unificada**. Nossa ferramenta visa **centralizar e traduzir conhecimento heterogêneo em um grafo canônico multi-camadas**, permitindo ao usuário **explorar analogias visualmente** através de um **heatmap de similaridade** com explicações claras das correspondências.

Público-alvo: Três personas exemplares serão beneficiadas:

- *Pesquisador interdisciplinar (Biologia ↔ Software)*: quer comparar conceitos de sistemas biológicos (ex.: *Homeostase*) com padrões de engenharia (*Autoscaling*) para **gerar insights inovadores** em pesquisa.
- *Engenheiro de software/arquitetura*: deseja **importar um repositório de código** ou arquitetura e compará-lo com sistemas conhecidos (ex.: comparar um micro-serviço com um modelo de organismo) para **identificar fragilidades ou inspirar melhorias** (ex.: sistema imune vs. mecanismos de segurança).
- *Analista organizacional (cibernética)*: quer mapear **estrutura organizacional e processos (VSM)** em comparação a outros sistemas (ex.: org vs. colônia de formigas) para **diagnosticar problemas** e propor ajustes baseados em analogias de sistemas viáveis.

Principais casos de uso: (Top 10)

- Comparar dois conceitos de alto nível:** O usuário seleciona Conceito A de Graph A e Conceito B de Graph B e visualiza instantaneamente **onde a analogia é mais forte** (células “quentes”) em diferentes camadas (ex.: *Apoptose* vs *TTL (Time-to-Live)*) mostrando alta similaridade na camada de “falha controlada”).
- Explorar subanalogias (drill-down):** Ao clicar numa célula quente no heatmap principal, a ferramenta exibe um **sub-heatmap** detalhando correspondências entre **subconceitos** (ex.: clicando *Homeostase* ↔ *Autoscaling*, ver *Feedback Negativo* ↔ *Backpressure*, *Redundância* ↔ *Multi-AZ Deployment*, etc.).
- Explicação de analogia:** Em qualquer correspondência selecionada, o painel “Explain” detalha **por que está quente**, incluindo os principais nós pareados, relações alinhadas e também **onde a analogia falha** (divergências).
- Importar grafos variados:** Usuário importa dados heterogêneos – um **JSON** de ontologia, um **trecho de texto** descritivo, ou um **repositório de código** – e a ferramenta converte tudo para o GraphSpec unificado, pronto para comparação.
- Comparar versões de um sistema:** O usuário carrega duas versões do mesmo grafo (ex.: arquitetura antes e depois de uma refatoração) e visualiza um “heatmap de diferenças” para ver **o que mudou** em termos de analogias internas ou com outro sistema (ex.: v1 vs v2 e como se aproximou de um sistema de referência).
- Ajuste fino e exploração guiada:** Usuário **ajusta pesos** dos critérios de similaridade (semântico,

estrutural, etc.) via controles avançados e imediatamente vê o **heatmap recalcular** (com caching) para explorar “e se” (ex.: dar mais peso a relações funcionais e ver emergir analogias diferentes).

7. Feedback e aprendizado: Em colaboração de equipe, usuários marcam manualmente certas analogias como “boas” ou “ruins”. O sistema registra esses feedbacks e **recalibra** ou sugere ajustes nos pesos, melhorando futuros resultados (aprendizado local, sem ML pesado inicialmente).

8. Busca e navegação rápida: A ferramenta oferece busca por nome/label de conceito; ao buscar “imun”, destaca no heatmap e lista a posição do conceito “Sistema Imune” no grafo selecionado, facilitando navegar grafos grandes.

9. Exportar e comunicar descobertas: Usuário exporta o heatmap em PNG (instantâneo visual de alta qualidade) ou SVG/JSON (para pós-processamento), incluindo as explicações principais, para usar em apresentações ou relatórios de pesquisa.

10. Uso offline com dados sensíveis: Instituições que trabalham com informações confidenciais (ex.: design proprietário de sistema) utilizam a ferramenta 100% offline e local, garantindo privacidade* e mantendo desempenho alto, sem depender de internet ou nuvem ¹.

O que NÃO é (fora de escopo): Esta ferramenta **não** é um serviço SaaS online nem colaborativo em tempo real – ela é local-first por design. **Não** é uma plataforma genérica de mineração de padrões ou aprendizado de máquina genérico: foca em **analogias estruturadas de grafos** (não em qualquer tipo de dado tabular, imagem, etc). **Não substitui especialistas humanos** – não “prova” que uma analogia é válida no mundo real (fornecendo sugestões e explicações, mas espera validação do usuário). **Não** é um motor de inferência ontológico completo (embora importe ontologias, não garantimos todos os raciocínios OWL). E **não** é um simples heatmap de telemetria de usuário ou ferramenta de BI – apesar de usar visualização de calor, seu contexto é conhecimento conceitual, não cliques de site ou métricas comerciais.

Diferenciais (Por que é único):

- **Multi-camadas com Drill-down:** Diferente de comparações simples de similaridade, nossa ferramenta permite **analisar analogias hierarquicamente**, indo do conceito macro até subestruturas, sem “quinhas” ou interrupções na exploração.
- **Explicabilidade Rigorosa:** Cada correspondência vem acompanhada de **justificativa** – os pontos de contato (ex.: ambos têm feedback negativo) e divergências são explicitados, aumentando a confiança do usuário e o aprendizado, em contraste a abordagens “caixa-preta”.
- **Desempenho & Visualização Top-Tier:** Renderização otimizada para **60fps** mesmo com grafos médios (~10k nós), usando Canvas/WebGL conforme necessário, com um cuidado especial em colormaps perceptualmente uniformes (ex.: Viridis) em vez de espectros enganosos ². A UI segue padrões de produto de primeira linha (tipografia nítida, spacing consistente, tooltips estáveis), tornando a exploração **prazerosa e fluida**.
- **Local-First & Privacidade:** Toda a funcionalidade roda localmente – o usuário pode operar offline sem qualquer dependência de nuvem. Os dados permanecem sob controle do usuário (não “trancados na nuvem”), reforçando privacidade e permitindo uso em cenários confidenciais ¹.
- **Arquitetura Plugável:** A ferramenta foi projetada desde o início para suportar **plugins de importação** de múltiplos formatos (texto livre, ontologias OWL, repositórios de código, etc.), sem “bagunçar” o núcleo. Isso significa que pode evoluir facilmente para novos domínios (ex.: importar modelos de ML, cenários físicos) adicionando módulos sem retrabalho pesado.
- **Cybernetics 2ª ordem (feedback adaptativo):** Inspirada em cibernetica de 2ª ordem, a ferramenta reconhece que ao prover métricas de analogia, pode influenciar as decisões do usuário. Incluímos mecanismos para **evitar Goodhart** e promover um **feedback loop saudável**, sinalizando quando um score

pode estar sendo otimizado de forma espúria ³. Essa abordagem consciente é rara em ferramentas analíticas tradicionais.

Métricas de sucesso: (quantitativas e mensuráveis)

- **Adoção e retenção:** Número de usuários ativos que integram a ferramenta em seus fluxos de trabalho (meta MVP: 10 usuários piloto; V1: 100+; V2: 500+).
- **Cobertura de casos de analogia:** Capacidade de ingerir diferentes tipos de input. Meta MVP: pelo menos 2 formatos (JSON grafo, Texto simples) funcionando; V1: + código; V2: + ontologias OWL; V3: + outros.
- **Desempenho (FPS e Tempo):** Heatmap principal renderizado a **60fps** em navegação panorâmica para grafos de $\sim 200 \times 200$ nós (meta MVP). Tempo de cálculo inicial de similaridade < 5 segundos para 200 nós \times 200 nós (meta). V2: escalar para 1000 \times 1000 nós com tiling e manter ao menos 30fps ao pan/zoom (com uso de WebGL se preciso).
- **Qualidade das analogias:** Após calibragem, pelo menos 80% das analogias marcadas “boas” pelos usuários especialistas devem aparecer nos top-N (N=3) células mais quentes. E <10% dos top-20 hotspots devem ser considerados “falsos” ou espúrios em avaliações cegas. (Métricas obtidas via feedback do usuário e testes internos).
- **Feedback loop utilizado:** % de analogias avaliadas pelos usuários. Meta: >30% das analogias exploradas recebem marcação (sinal de engajamento) no MVP; aumentar para >50% no V2 com UX otimizada de feedback.
- **Nenhum vazamento de dados:** 0 incidentes de dados enviados externamente. Podemos validar monitorando que a aplicação não faz requisições web quando em modo offline (testes de rede).
- **Estabilidade e ausência de crashes:** MTBF (tempo médio entre falhas) alto – idealmente nenhum crash crítico relatado durante um mês de uso contínuo nos cenários de teste.

Restrições essenciais: O produto deve ser **100% local-first e offline por padrão** – todas as operações ocorrem localmente e os dados persistem localmente (SQLite, arquivos). Se houver componentes opcionais de ML (ex.: embeddings), eles devem ser baixáveis e executados localmente, nunca enviando dados para terceiros sem consentimento explícito. O desempenho é crucial: a UI e visualização têm orçamento estrito (alvo 60fps) e cálculos de similaridade devem evitar travar a interface (uso de threads ou async). Privacidade e segurança: nenhuma dependência de nuvem; se sincronização ou colaboração for adicionada no futuro, será opt-in e criptografada. Restrições de hardware: assumir **CPU only** a princípio (não exigir GPU CUDA, embora use WebGL no browser), suportar plataformas desktop comuns. Suportar multiplataforma via web (e eventualmente desktop wrapper), mantendo **persistência de dados local** indiferente do ambiente.

North Star (valor em 30s): Um usuário abre a ferramenta pela primeira vez, carrega dois conjuntos de conceitos que lhe são familiares (ex.: um grafo da biologia celular e outro da arquitetura de micro-serviços). Em meio minuto, ele vê um mapa de calor onde uma região se acende em vermelho intenso correlacionando “Apoptose” (morte celular programada) com “Graceful Shutdown/TTL” (encerramento programado de serviço). Intrigado, ele clica no ponto e imediatamente surge uma explicação: ambos são mecanismos de falha controlada com remoção segura de partes do sistema, suportados por feedbacks e sinais de término. O usuário sorri ao ver confirmada uma analogia que antes estava apenas intuída. **Essa validação instantânea e visual de uma hipótese cross-domain é a prova de valor** – em 30 segundos a ferramenta ofereceu um insight açãoável e explicável, algo que demoraria dias de pesquisa manual.

B) User Stories + Jobs-to-be-Done

A seguir, apresentam-se histórias de usuário representativas, cobrindo necessidades e objetivos (jobs) das personas. Cada história inclui critérios de aceitação, dados necessários e possíveis riscos.

História 1: Como pesquisador interdisciplinar, quero comparar dois conceitos de alto nível de domínios diferentes para identificar se existe uma analogia forte entre eles em termos de função e estrutura.

- **Critérios de Aceitação:** Dado dois conceitos selecionados (um em cada grafo carregado), o sistema calcula a similaridade e destaca a célula correspondente no heatmap principal. Se a analogia estiver entre as top-N mais quentes, ela deve aparecer em destaque (ex.: cor forte, ou listagem de "Top analogias"). O usuário deve conseguir clicar na célula e visualizar no painel de explicação pelo menos **3 evidências** de correspondência (ex.: tags funcionais iguais, relação similar com subcomponente, pontuação semântica alta) e possíveis diferenças.

- **Dados Necessários:** Dois grafos carregados contendo os conceitos em questão. Cada conceito com seus subconceitos e relações relevantes já importados no GraphSpec.

- **Riscos:** Se os grafos estiverem incompletos ou os conceitos não tiverem contextos suficientes, a analogia pode não emergir (falso negativo). Risco de **excesso de confiança**: usuário pode interpretar um score moderado como "não existe analogia" quando pode haver, exigindo UI clara sobre incerteza.

História 2: Como pesquisador, quero explorar as camadas de analogia (drill-down) para entender quais subconceitos específicos sustentam a analogia entre dois conceitos principais.

- **Critérios:** Na interface, após visualizar a analogia macro (ex.: *Homeostase ↔ Autoscaling*), o usuário aciona um drill-down (via clique ou botão) e o sistema exibe um sub-heatmap onde os eixos são os **subcomponentes** de Homeostase (ex.: *Feedback Negativo, Feedback Positivo, Sinalização Hormonal*) vs. de Autoscaling (ex.: *Metric Monitoring, Scaling Policy, Cooldown Timer*). O sub-heatmap deve recalcular similaridades restritas a essas subestruturas. Critério de sucesso: as correspondências relevantes aparecem claras (ex.: *Feedback Negativo* bem alinhado com *Scaling Policy*, etc.) e o usuário pode navegar de volta à visão anterior facilmente.

- **Dados:** O grafo deve ter hierarquia (nós filhos relacionados por arestas `part_of` ou similar). Precisamos das relações de composição para construir os subgrafos.

- **Riscos:** Grafos muito assimétricos (um conceito com muitos subconceitos e outro com poucos) podem tornar a visualização confusa ou a analogia pobre. Deve-se mitigar mostrando também divergências ("conceito A tem 5 partes vs. conceito B tem 2 partes"). Risco de **explosão combinatória** se o usuário tentar drill-down de muitos pares simultaneamente – limitar interações a um par por vez.

História 3: Como engenheiro de software, quero importar um repositório de código (em Python) e automaticamente gerar um grafo de componentes e dependências, para então compará-lo a um grafo de sistemas biológicos.

- **Critérios:** O usuário executa uma ação de import (via UI ou CLI) apontando para a pasta do repo. O sistema aplica o plugin `import_repo_python`, extrai módulos, classes, funções como nós, e dependências (ex.: "importa" ou "chama função X") como arestas. Critério de aceitação: após import, um grafo estruturado aparece na lista de grafos com meta-informações (nome, origem "repo import", nº de nós). O usuário pode então selecionar esse grafo e compará-lo a outro (ex.: biologia). Deve ser possível visualizar conceitos de código (ex.: *module circuit_breaker.py*) alinhados com conceitos biológicos (*hormesis mechanism?*) se houver analogias.

- **Dados:** Código fonte disponível localmente. Dependemos de conseguir parsear (AST) o código, portanto o repo deve ser sintaticamente correto. Também úteis convenções de projeto para inferir camadas (ex.: pasta

"services/" = subsistema).

- *Riscos*: Repositórios grandes podem gerar grafos enormes (milhares de nós); risco de performance ou ruído (muitos nós triviais). Mitigação: plugin deve permitir filtragem (ex.: ignorar arquivos de teste, ou incluir só componentes acima de certo tamanho) e/ou agregar automaticamente. Outro risco é **classificação incorreta** de elementos (ex.: confundir classes utilitárias com componentes core) – isso afetaria analogias funcionais.

História 4: Como engenheiro/analista, quero anotar analogias como “boas” ou “ruins” e ver o sistema ajustar a relevância dessas sugestões ao longo do tempo.

- *Critérios*: Na UI, ao visualizar uma correspondência (célula no heatmap), o usuário clica em “Boa analogia” (👍) ou “Ruim” (👎). A aceitação: o sistema registra essa avaliação persistentemente (ex.: em um banco local), e opcionalmente recalcula as pontuações ajustando pesos ou marcando aquele par de nós para evitar/propor futuras correspondências. Em interações futuras, analogias marcadas como “ruim” devem aparecer atenuadas ou ser menos priorizadas, enquanto “boas” poderiam subir de rank. Uma mudança notável (mas não abrupta) nos scores ou ordenação de hotspots após várias marcações seria critério de sucesso, indicando aprendizado local.

- *Dados*: Armazenamento local para feedback (associado ao par de IDs do GraphSpec ou características delas). Um algoritmo de ajuste de pesos ou penalização de pares.

- *Riscos*: Pouco feedback pode levar a **sobreajuste** se o sistema supervalorizar poucas marcações. Também risco de **vies do usuário**: se o usuário só marca exemplos óbvios, o sistema pode aprender a ignorar analogias sutis mas válidas. Mitigar garantindo que o feedback influencia de forma gradual e explicável (talvez mostrando “peso ajustado via feedback” nas explicações).

História 5: Como usuário, quero entender *por que* o sistema considerou duas coisas análogas – quero uma explicação verificável.

- *Critérios*: Dado um par (nó A em grafo X, nó B em grafo Y) que aparece com score alto, o usuário abre a aba/painel “Explain”. Critério de aceitação: o painel lista separadamente: (a) **Principais mapeamentos de nós** (ex.: A→B direto, A1 (filho de A) → B2 (filho de B), etc.), com seus pesos; (b) **Relações suportantes** – ex.: “A causa C e B causa D, e C↔D foi analogia forte” indicando preservação de estrutura; (c) **Divergências** – ex.: “A é cíclico enquanto B é linear”, ou “A opera em escala de milissegundos vs B em dias”. O usuário deve conseguir rastrear no grafo ou texto original essas evidências (por ex., mostrar IDs ou trechos se texto). Sucesso é quando o usuário consegue explicar a analogia a outra pessoa usando essas informações, sem precisar adivinhar.

- *Dados*: O GraphSpec deve ter informações suficientes (tags, tipos de aresta, possivelmente descrições textuais) para gerar explicações. Precisamos de um repositório de frases modelo (ex.: para descrever cada dimensão).

- *Riscos*: Se os grafos não tiverem metadados ou contextos, a explicação pode ficar superficial (“scores numéricos sem justificativa clara”). Mitigação: incentivar preenchimento de tags/domínios no import; no pior caso, explicar via similaridade textual (“95% de overlap de termos”) – mas isso pode ser pouco interpretável.

História 6: Como usuário avançado, quero ajustar parâmetros de cálculo (ex.: pesos das dimensões de similaridade, profundidade de busca) para testar diferentes cenários de analogia.

- *Critérios*: A interface oferece um modo “Avançado” com controles (sliders, campos numéricos) para os pesos de cada dimensão (Semântico, Estrutural, Funcional, Temporal). O usuário modifica, por exemplo, Semântico de 0.4 para 0.2 e Estrutural de 0.3 para 0.5. Ao aplicar, o sistema recalcula rapidamente (usando cache onde possível) e atualiza o heatmap. Aceitação: o usuário vê mudanças coerentes – e.g., analogias

que antes eram quentes puramente por similaridade de nome agora esfriam se reduzido o peso semântico. Além disso, deve haver um campo para limitar *profundidade* (layer) ou *k-hop* vizinhos considerados; ao alterá-lo (ex.: considerar até 2 hops em vez de 3), o heatmap recalcula dentro de segundos.

- *Dados*: Os scores e caches precisam ser re-computáveis parametricamente. Dados dos grafos já carregados; potencialmente caches indexados por combinações de parâmetros? (MVP recalcula do zero ou parcial.)

- *Riscos*: Expor muitos parâmetros pode confundir usuários não técnicos – mitigar ocultando por padrão e oferecendo presets. Risco de combinação inválida (ex.: todos pesos zero) – UI deve prevenir ou dar erro claro. Performance: recalcular tudo toda vez pode ser pesado – preferível cachear componentes.

História 7: Como analista de sistemas organizacionais (VSM), quero importar uma ontologia (OWL/RDF) de termos de gestão e compará-la com uma ontologia de biologia para descobrir analogias entre funções organizacionais e funções biológicas.

- *Critérios*: Via plugin `import_ontology`, o usuário carrega um arquivo OWL. A ferramenta converte classes em nós e propriedades em arestas (ex.: *Departamento* is_a *Sistema*, *Departamento* hasRole *S3...*). Aceitação: o grafo importado mantém a hierarquia (campos layer indicam classes vs subclasses) e podemos visualizar, por exemplo, *Função de Gestão S3* alinhada a *Homeostase* do grafo biológico se existir base análoga.

- *Dados*: Arquivo OWL ou URL local. Necessário suporte a RDF parsing e possivelmente mapeamento de predicados OWL para tipos de aresta (subClassOf -> is_a, objectProperty -> custom edge).

- *Riscos*: Ontologias grandes (centenas de classes) – risco de performance e ruído; mitigar pedindo ao usuário focar em uma sub-ontologia ou limitando profundidade. Também podem haver muitos detalhes lógicos (restrições) que não importamos – deve ficar claro que a semântica formal completa não será usada além do grafo estrutural básico.

História 8: Como time de desenvolvimento local-first, queremos integrar a ferramenta ao nosso pipeline interno (via CLI ou API local) para gerar heatmaps automaticamente a partir de cada novo desenho de arquitetura e verificar similaridade com sistemas anteriores.

- *Critérios*: Deve existir uma interface de automação: por exemplo, um CLI `analogyttool import file.json` que retorna sucesso e armazena grafo, ou `analogyttool compare A B --out result.png` que gera um PNG do heatmap A vs B. Critério de aceitação: o time consegue escrever um script que, a cada commit de arquitetura (GraphSpec JSON), roda a ferramenta headlessly e produz artefatos (imagens, JSON de scores) que são arquivados. A API local (ex.: endpoints HTTP na porta localhost) também deve permitir requisições simples para integrar com outras apps.

- *Dados*: Grafos em arquivo ou identificadores, acesso de linha de comando ou local web service.

- *Riscos*: Segurança – CLI/API deve ser local apenas, para não abrir brecha. Versatilidade – riscos de compatibilidade em diferentes OS ou ambientes headless (precisamos garantir que a ferramenta pode rodar sem UI para esses casos).

História 9: Como usuário, quero salvar meu progresso e resultados (grafos importados, analogias marcadas, configurações) e poder fechar e reabrir a ferramenta continuando de onde parei.

- *Critérios*: O aplicativo deve salvar localmente o estado: grafos persistem (em disco/SQLite), incluindo versões importadas; feedback do usuário e ajustes de peso permanecem entre sessões. Ao reabrir, os últimos grafos comparados poderiam ser reabertos automaticamente com o heatmap em cache (marcado como possivelmente *stale* se a data do grafo mudou). Aceitação: o usuário fecha e abre a ferramenta, verifica que não precisa reimportar tudo nem reconfigurar pesos – tudo relevante está lá (talvez via opção

de “restaurar último projeto”).

- *Dados*: Necessário persistir GraphSpec JSONs (provavelmente como arquivos ou em DB), as configurações de usuário (JSON config), cache de resultados (SQLite ou arquivos), e feedback.

- *Riscos*: Corrupção de cache/DB – mitigar usando SQLite robustamente e realizando backups ou transações. Versão de schema: se atualizarmos o GraphSpec entre versões, migrar dados antigos.

História 10: Como analista preocupado com riscos, quero ser alertado quando uma analogia “quente” pode ser enganosa ou de alto risco (ex.: baseada apenas em nome similar mas sem substância).

- *Critérios*: Quando uma célula tem score alto porém com indicadores de incerteza (ex.: dimensão semântica muito alta mas estrutural muito baixa), o sistema marca visualmente (ícone Δ ou hachuras na célula) e, ao passar o mouse, exibe um tooltip “Possível analogia espúria – verifique estrutura”. Aceitação: Em testes, uma analogia conhecida como falsa (ex.: dois conceitos com nome parecido mas contextos distintos) é sinalizada adequadamente pelo sistema. Além disso, no painel Explain de tal caso deve haver uma seção “Risco” apontando o motivo (e.g. “score alto dominado por semântica, relações não alinhadas”).

- *Dados*: Lógica de cálculo de confiança/risco dentro do algoritmo de hotness (ex.: variância entre dimensões, ou flag de extrapolação).

- *Riscos*: Alertas demais podem irritar usuário (falso positivo de risco) – calibragem é crucial. Precisamos evitar *cry wolf*: só alertar quando realmente relevante.

História 11: Como usuário curioso, quero pesquisar um termo dentro dos grafos carregados para localizar conceitos específicos e entender seu contexto antes de compará-los.

- *Critérios*: Haver um campo de busca global. Ao digitar, por ex, “feedback”, o sistema lista nós correspondentes nos grafos (ex.: *Feedback Negativo* em grafo Bio, *Feedback Loop* em grafo Software) e destaca suas posições no heatmap se um comparativo estiver ativo (por exemplo, destaca linha/coluna correspondente). Se nenhum heatmap aberto, talvez mostra as ocorrências e permite selecionar dois para comparar. Aceitação: ao selecionar um resultado da busca, a interface responde rapidamente (sub-500ms para lista) e facilita o usuário tomar ação (ex.: “Adicionar esse nó à seleção de comparação” ou “ir para grafo X e ver detalhes do nó”).

- *Dados*: Índice invertido de labels e possivelmente descrições, armazenado local (SQLite full-text search?). Grafos já em DB.

- *Riscos*: Grafos grandes podem gerar muitos resultados – UI deve paginar ou priorizar melhor match. Precisamos garantir que a busca não **trave** durante digitação (usar debounce e thread/worker).

História 12: Como gerente de produto (da equipe desenvolvedora da ferramenta), quero um registro de uso e performance (telemetria local) para validar se a experiência está dentro dos padrões (FPS, tempos) e onde melhorar.

- *Critérios*: A aplicação mantém logs locais (opt-in se necessário) com eventos como “tempo de import X ms”, “tempo de cálculo heatmap Y ms”, “FPS médio durante pan: Z”. O gerente pode acessar um painel ou arquivo de log consolidado. Aceitação: Após uma sessão de uso, existem logs que mostram que performance esteve ok (ou não). Exemplo: log mostrando que no dataset de 500 nós, cálculo levou 3s (<5s meta) e FPS manteve ~58-60 em interações – satisfazendo critérios.

- *Dados*: Coleção de métricas instrumentadas no código (timers, frame counters) e escritas localmente.

- *Riscos*: Respeitar privacidade – não coletar dados dos grafos do usuário, apenas métricas genéricas. Risco de overhead – logging em excesso pode afetar performance, mitigar com amostragem ou desligar logs detalhados por padrão.

História 13: Como analista, quero comparar dois estados do mesmo sistema (antes vs depois de mudança) para ver como as analogias internas ou com outro sistema evoluíram.

- **Critérios:** Usuário carrega grafo Versão 1 e Versão 2 (por ex., antes e depois de adicionar um módulo X). Seleciona uma opção “Comparar versões” possivelmente escolhendo um terceiro grafo de referência para analogia ou comparando V1 vs V2 diretamente. Aceitação: o sistema destaca diferenças – por exemplo, heatmap V1 vs V2 pode ter célula da funcionalidade X vs X (mesma entidade) mas com “temperatura” diferente indicando mudança de contexto; ou comparando cada versão com um mesmo Graph B, mostrar que V2 teve analogia ligeiramente mais forte com algo desejável. Deve haver visualização clara dessas diferenças (ex.: outline ou delta).
- **Dados:** Versões identificáveis (GraphSpec com version metadata). Um mecanismo de comparação (diferença nas arestas/no conteúdo).
- **Riscos:** Pode ser complexo apresentar 3 dimensões (Graph A vs Graph B vs Graph B') – possivelmente fixar comparação a duas a cada vez. Também risco de ruído se pequenas mudanças resultam em muitas pequenas variações – necessitamos possivelmente um filtro para mostrar só mudanças acima de certo limiar.

História 14: Como especialista de domínio, quero incorporar conhecimento manualmente ajustando ou criando grafos e refinando-os, e ver esses refinamentos refletidos imediatamente nas analogias.

- **Critérios:** Fornecer modo de edição ou pelo menos import de JSON editado. Aceitação: o usuário edita o GraphSpec JSON (por ex., adiciona uma aresta que faltava, ou corrige uma label) e salva. A ferramenta detecta a mudança (hash ou timestamp) e marca o heatmap como *desatualizado* (stale), convidando a recalcular. Ao recalcular, resultados atualizam (ex.: analogia antes fraca agora fica mais forte pois aquela relação adicionada era crucial). Alternativamente, um mini editor de nós/arestas na UI para ajustes rápidos seria sucesso (MVP pode ser manual JSON reload).
- **Dados:** GraphSpec serializado facilmente, e um watcher de arquivo ou manual “reimport”.
- **Riscos:** Edição manual pode introduzir erros JSON – precisamos validação e mensagens claras. Sincronização: se usuário edita enquanto heatmap aberto, evitar condições de corrida.

História 15: Como usuário preocupado com privacidade, quero garantir que a ferramenta nunca envie meus dados ou resultados para a internet, a menos que eu explicitamente exporte algo.

- **Critérios:** Código e arquitetura não fazem fetch/requests externas sem comando do usuário. Podem ser realizadas verificações (ex.: user roda monitor de rede e não vê tráfego) ¹. O software deve documentar esse compromisso. Aceitação: revisão de código ou testes mostram ausência de endpoints externos e uso apenas de recursos locais.
- **Dados:** N/A (é sobre comportamento).
- **Riscos:** Dependências de bibliotecas que “telefonam para casa” – mitigado escolhendo libs offline ou desligando telemetria delas. Usuário deve ter confiança, possivelmente providenciar opção “modo avião forçado” dentro do app.

(As histórias acima cobrem desde a exploração básica de analogias até integrações avançadas e preocupações de privacidade/qualidade. Elas guiam o desenvolvimento incremental do produto, garantindo que cada recurso atenda a uma necessidade real do usuário.)

C) Modelo Canônico - GraphSpec (JSON)

Para unificar entradas diversas (texto, ontologia, código) em uma base comum, definimos o **GraphSpec**, um formato canônico em JSON para representar grafos conceituais multilayer. O GraphSpec impõe consistência

de identificação, tipagem e estrutura, servindo como língua franca entre importadores, motor de analogia e visualização.

Elementos principais:

- **Nó (Node):** representa um conceito ou entidade. Atributos:
 - `id` (string): Identificador único no grafo (ex.: "bio-Homeostase-01"). Deve ser único por grafo e preferencialmente estável (usado para mapear correspondências e feedback).
 - `label` (string): Nome legível do conceito (pode incluir espaços, acentos). Ex.: "Homeostase".
Normalização: manter consistência de idioma e nomenclatura dentro do grafo; caracteres especiais em labels são permitidos, mas `id` usa um subconjunto seguro (ex.: slug).
 - `domain` (string): Domínio ou categoria do conceito, ex.: "biologia", "software", "organizacional". Pode ser usado para estilização ou restrição de comparações (ex.: não comparar diretamente nós de domínios totalmente díspares sem contexto).
 - `layer` (int): nível de profundidade/camada do nó dentro de seu grafo. Ex.: 0 para conceitos de topo (gerais), 1 para subconceitos imediatos, etc. Camadas podem derivar de relações hierárquicas (ex.: arestas `is_a` ou `part_of`). Esse campo permite filtrar comparações por nível e guiar drill-down (nós de layer=1 são comparados no sub-heatmap quando seus pais de layer=0 foram comparados).
 - `tags` (array[string]): marcadores opcionais qualificando o papel ou características do nó. Exemplos de tags: "VSM:S1" (identifica papel do Viable System Model, Sistema 1), "control_loop:feedback_negative", "timescale:days", "failure_mode:graceful", "pattern" etc. Essas tags padronizadas enriquecem as dimensões de comparação (ex.: dois nós com tag `failure_mode:graceful` indicam ambos lidam com falha suave). Múltiplas tags podem existir por nó.
 - `embedding` (array[float], opcional): vetor de embedding semântico (ex.: 128 dimensões) representando o conceito em um espaço latente. Preenchido se disponível (ex.: gerado offline via modelo tipo Word2Vec ou S-BERT). Usado para cálculo semântico refinado. Se não presente, a comparação semântica recorre a métodos lexicais.
- Outros possíveis campos: `description` (string, opcional - texto descritivo do conceito), `properties` (mapa chave->valor livre para extensões ou metadados).
- **Aresta (Edge):** representa relação entre dois nós. Atributos:
 - `source` (string): id do nó origem.
 - `target` (string): id do nó destino.
 - `type` (string): tipo da relação, definindo sua semântica. Exemplos de tipos padrão: "is_a" (A é um tipo de B), "part_of" (A é parte de B), "depends_on" (A depende de B), "causes" (A causa B), "regulates" (A regula B), "analogous_to" (marcação explícita de analogia conhecida), etc. Esses tipos idealmente seguem padrões ontológicos quando possível (ex.: relações OWL básicas, ou propriedades definidas pelo plugin importador).
 - `direction` (string, opcional se já implicado): "directed" ou "undirected". Por padrão, relações como `causes` são dirigidas, enquanto `analogous_to` poderia ser não-dirigida.
 - `weight` (number, opcional): peso ou força da relação no contexto do grafo original. Ex.: 1.0 padrão, ou uma medida (0-1) de confiança ou intensidade. Importadores podem atribuir pesos (ex.: grau de associação textual, frequência de chamada de função, etc.).

- `evidence` (string, opcional): referência de evidência da relação – pode ser uma citação, URL, ou trecho de texto que justifica a existência dessa aresta (útil especialmente em `import_text` ou `import_ontology`). Ex.: "Literature: DOI 10.1038/s41586-020" ou "Source code: moduleX.py line 40".

- **Meta do Grafo:** informações sobre o grafo como um todo, armazenadas no nível raiz do JSON ou em um campo separado, por exemplo:

- `origin` (string): origem do grafo (ex.: `"imported_from": "repo.github.com/user/proj"` ou `"source": "custom_manual"` ou `"ontology: myonto.owl"`).
- `name` (string): nome/título do grafo (curto).
- `description` (string, opcional): descrição do escopo do grafo.
- `version` (string ou int): versão do grafo. Pode ser número de versão ou hash. Útil para controle de versões e detecção de mudanças.
- `date` (string/timestamp): data de criação/última modificação.
- `confidence` (number, opcional): confiança geral no grafo (ex.: uma pontuação de qualidade atribuída pelo importador, 1.0 = alta confiança nos dados).
- `license` (string, opcional): licença ou restrição de uso dos dados (ex.: "CC-BY" se os dados originários têm licença).
- `schema_version` (string): versão do formato GraphSpec usado, garantindo compatibilidade futura (ex.: "1.0").

Representação de camadas e subárvores: A relação de camada pode ser inferida através das arestas (ex.: cada `part_of` ou `is_a` aumenta layer do nó filho em +1 em relação ao pai). Entretanto, para facilitar o acesso direto, armazenamos o atributo `layer` em cada nó. Subárvores podem ser extraídas pegando todos nós descendentes de um nó de topo: por exemplo, para drill-down, quando o usuário clica numa analogia envolvendo nó A (layer 0) ↔ B (layer 0), pegamos todos nós de layer 1 que são descendentes de A e todos de layer 1 descendentes de B (via arestas `part_of` ou `is_a`), formando subgrafos para análise. A GraphSpec não impõe limite rígido de profundidade, mas recomenda-se manter hierarquias razoáveis (ex.: 0 = sistema, 1 = subsistema, 2 = componentes, 3 = subcomponentes etc., evitando dezenas de níveis sem necessidade).

Normalização de IDs e labels: IDs devem ser únicos e preferencialmente deriváveis do label de forma estável (ex.: lower snake case, prefixo de domínio). Importadores podem gerar IDs determinísticos (talvez um hash do caminho hierárquico). Labels podem conter maiúsculas, acentos, mas para comparações semânticas textuais, pode-se usar versões normalizadas (sem acento, minúscula) internamente – sem alterar o label exibido. Uma convenção: no JSON, manter `label` original e ter um campo oculto ou no índice textual com versão normalizada para busca/semântica.

Versionamento de grafos: Cada grafo possui seu `version` e `date`. Para permitir histórico e “comparar versões” (user story 13), a ferramenta pode armazenar múltiplos GraphSpecs com mesmo `name` mas `version` diferente. O sistema de cache deve usar `(grafoA_id, versionA, grafoB_id, versionB, parâmetros)` como chave para evitar confusões. Podemos implementar diffs mostrando mudanças entre versões: ex., comparando GraphSpec v1 vs v2 – arestas novas, removidas, tags alteradas – como uma lista. (Embora a visualização de diferenças de analogia entre versões seja um recurso avançado, o versionamento no modelo é o primeiro passo para isso.)

Exemplo mínimo – Grafo de Biologia (trecho simplificado, ~10 nós):

```
{
  "name": "Biologia - Sistema Celular",
  "origin": "manual",
  "domain": "biologia",
  "version": "1.0",
  "nodes": [
    { "id": "bio1", "label": "Homeostase", "domain": "biologia", "layer": 0,
      "tags": ["processo", "VSM:S5", "control_loop:feedback_negativo"] },
    { "id": "bio2", "label": "Apoptose", "domain": "biologia", "layer": 1,
      "tags": ["mecanismo", "failure_mode:graceful"], "embedding": [0.12,
      0.03, [ ]] },
    { "id": "bio3", "label": "Necrose", "domain": "biologia", "layer": 1,
      "tags": ["mecanismo", "failure_mode:descontrolado"] },
    { "id": "bio4", "label": "Feedback Negativo", "domain": "biologia",
    "layer": 1,
      "tags": ["mecanismo", "control_loop:feedback_negativo"] },
    { "id": "bio5", "label": "Sistema Imune", "domain": "biologia", "layer": 0,
      "tags": ["estrutura", "VSM:S1"] },
    { "id": "bio6", "label": "Inflamação", "domain": "biologia", "layer": 1,
      "tags": ["resposta", "failure_mode:tradeoff"] }
    // ... outros nós ...
  ],
  "edges": [
    { "source": "bio2", "target": "bio1", "type": "part_of" },
    { "source": "bio3", "target": "bio1", "type": "part_of" },
    { "source": "bio4", "target": "bio1", "type": "supports", "evidence":
    "observação: regula homeostase" },
    { "source": "bio2", "target": "bio5", "type": "inhibits", "evidence":
    "literature: DOI 12345" },
    { "source": "bio6", "target": "bio5", "type": "part_of" }
    // ... ex.: Apoptose faz parte da Homeostase como mecanismo de controle
    saudável,
    // Necrose também relacionada a Homeostase porém de forma descontrolada,
    // Feedback Negativo sustenta Homeostase,
    // Sistema Imune tem subsistema Inflamação etc ...
  ]
}
```

(No exemplo acima: "Homeostase" (layer 0) contém subconceitos "Apoptose" e "Necrose" (ambos layer 1) como mecanismos de controle de população celular - apoptose sendo controlada (graceful) e necrose descontrolada. "Feedback Negativo" (layer 1) suporta Homeostase, provendo estabilidade. "Sistema Imune" (layer 0) tem "Inflamação" (layer 1) como parte. Tags indicam papéis: VSM:S5 sugere Homeostase análogo a função de política/gestão (Sistema 5) no VSM; Sistema Imune marcado como VSM:S1 indicando operação básica. Esses tags e relações ajudarão a encontrar correspondências em outro grafo.)

Exemplo mínimo – Grafo de Software (trecho ~10 nós):

```
{  
  "name": "Arquitetura Microserviços - Resiliência",  
  "origin": "import_repo_python",  
  "domain": "software",  
  "version": "0.1",  
  "nodes": [  
    { "id": "sw1", "label": "Autoscaling", "domain": "software", "layer": 0,  
      "tags": ["mechanism", "control_loop:feedback_negativo",  
"timescale:mins"] },  
    { "id": "sw2", "label": "Circuit Breaker", "domain": "software", "layer": 0,  
      "tags": ["pattern", "failure_mode:graceful"] },  
    { "id": "sw3", "label": "Restart/TTL", "domain": "software", "layer": 1,  
      "tags": ["mechanism", "failure_mode:graceful"] },  
    { "id": "sw4", "label": "Bulkhead (Isolamento)", "domain": "software",  
"layer": 0,  
      "tags": ["pattern", "failure_mode:compartimentalização"] },  
    { "id": "sw5", "label": "Monitor de Métricas", "domain": "software",  
"layer": 1,  
      "tags": ["component", "sensor"] }  
    // ... outros nós ...  
  ],  
  "edges": [  
    { "source": "sw3", "target": "sw2", "type": "part_of", "evidence": "code:  
class TTL inside CircuitBreaker" },  
    { "source": "sw3", "target": "sw1", "type": "part_of", "evidence": "config:  
TTL part of scaling policy" },  
    { "source": "sw5", "target": "sw1", "type": "supports", "weight": 0.9 },  
    { "source": "sw4", "target": "sw2", "type": "parallel_to", "evidence":  
"pattern: both isolate failures differently" }  
    // ... ex.: Restart/TTL é componente tanto do Circuit Breaker (um TTL pode  
ser usado ao desligar circuito)  
    // quanto do Autoscaling (encerrando instâncias velhas), atuando como  
"apoptose" do software.  
    // Monitor de Métricas suporta Autoscaling (feedback loop).  
    // Bulkhead e Circuit Breaker em paralelo como padrões complementares de  
isolamento de falha.  
  ]  
}
```

(No grafo de software: “Autoscaling” é um mecanismo de controle com feedback (tags indicam semelhança a feedback negativo). “Circuit Breaker” é um padrão de falha controlada (degradação graciosa). “Restart/TTL” é um subconceito (layer 1) presente dentro de Circuit Breaker e Autoscaling – representando a ideia de matar instâncias (apoptose do mundo software). “Bulkhead” é isolamento de falhas (compartimentalização). “Monitor de Métricas” (layer 1) é parte do loop de Autoscaling (sensor que alimenta o feedback).)

Esses exemplos ilustram a estrutura do GraphSpec. Ambos grafos têm conceitos com tags e relações análogas, o que espera-se resultar em hotspots: ex. *Apoptose* (*bio*) \leftrightarrow *Restart/TTL* (*sw*) (ambos *failure_mode:graceful*), *Homeostase* (*bio*) \leftrightarrow *Autoscaling* (*sw*) (ambos loops de controle com feedback negativo), *Compartimentalização* (*bio*, *implícito em órgãos*) \leftrightarrow *Bulkhead* (*sw*), *Sistema Imune* (*bio*, *S1*) \leftrightarrow *Observabilidade/Monitoramento* (*sw, sensor*) etc.

Validação e integridade: Uma vez importado/gerado, um GraphSpec deve passar por validação interna: checar duplicação de IDs, circularidade estranha (ciclos em `is_a` hierarquia devem ser evitados), tipos de aresta reconhecidos (ou registrar novos), formato de embeddings correto, etc. Logs de importação (discutidos em seção F) registrarão se algo foi ajustado (ex.: IDs normalizados, aresta ignorada). Assim garantimos que o core opere em cima de dados consistentes.

D) Motor de “Hotness” – Cálculo de Similaridade de Analogias

O “Hotness Score” quantifica o quanto forte é a analogia entre partes de dois grafos. Ele é concebido de forma **vetorial explicável**, compondo múltiplas dimensões de correspondência em um único score agregado. Isso permite granularidade na análise e evita depender de um único critério (reduz risco de Goodhart).

Dimensões de Similaridade (vetor de features): Propomos inicialmente 4 dimensões principais, cada uma resultando em uma pontuação normalizada [0,1] para um par de elementos (nós ou subestruturas) entre Grafo A e Grafo B:

1. **Semântica (Conceitual):** Mede similaridade de significado entre os conceitos em si (nomes, descrições e embeddings). Combina comparação de embeddings com fallback lexical.
2. **Como calcular:** Se ambos nós possuem `embedding`, calcula-se a similaridade de cosseno entre os vetores. Caso contrário, realiza-se comparações de string e contexto: ex., TF-IDF ou bag-of-words das descrições, ou até comparações de sub-palavras (levenshtein, trigramas) se for pouca informação. Normaliza-se o valor (0 = totalmente diferentes, 1 = virtualmente o mesmo termo).
3. **Exemplo:** “Apoptose” vs “TTL/Restart”: semântica textual baixa (palavras distintas), mas se houver embedding treinado em literatura, elas podem aparecer próximas por contexto (idealmente capture que ambas são mecanismos programados de término).
4. **Explicação:** Uma pontuação alta aqui indica “esses conceitos são semanticamente parecidos ou frequentemente associados”. Virá acompanhado de evidências como “compartilham termos X” ou “embedding proximidade = 0.87”.
5. **Estrutural/Relacional:** Compara as **relações e vizinhos** em torno dos dois nós, ou compara padrões de subgrafo.
6. **Como calcular:** Para um par de nós (A em grafo1, B em grafo2), considera-se suas vizinhanças (1-hop, possivelmente até k-hop controlado pelo parâmetro “k-hop”). Cria-se uma assinatura estrutural: por exemplo, conjuntos de tipos de aresta e graus (quantos filhos `part_of`, quantos pais `is_a`, etc.), ou mesmo um pequeno grafo de vizinhança. Faz-se o matching entre vizinhanças: qual fração das relações de A tem correspondentes análogas nas relações de B? (Para isso, precisam-se considerar os vizinhos de A e B e tentar alinhá-los também – uma recursão controlada).

7. Simplificando no MVP: calcula-se similaridade de *vetor de graus por tipo de relação*. Ex.: Nó A tem 2 `part_of` filhos e 1 `causes`, Nó B tem 3 `part_of` filhos e 1 `causes` – relativamente parecido. Ou criar vetores contando estruturas (grau, clustering, centralidade) e comparar.
8. *Exemplo:* Homeostase (bio) tem 2 subcomponentes (Apoptose, Necrose) e suporta integridade; Autoscaling (sw) tem subcomponentes (monitor, policy, TTL). Estrutural: ambos têm múltiplos componentes e fazem parte de loops – pontuação moderada/alta.
9. *Explicação:* Uma nota alta indica “os papéis que este conceito exerce na rede de relações é similar”. Evidências: “Ambos possuem ~2 mecanismos internos (`part_of`), e ambos são influenciados por feedback loops” etc.
10. **Funcional/De Papel (Role):** Verifica se os conceitos cumprem funções análogas no *sentido cibernético* ou de design – aqui entram as tags e padrões de papel (ex.: VSM roles, control loops, failure modes).
11. *Como calcular:* Compara as listas de tags de A e B. Se compartilham tags raras (ex.: ambos marcados `failure_mode:graceful` e `control_loop:feedback_negativo`), isso pesa bastante. Também verifica padrões: ex., se A é um nó de layer 0 com vários mecanismos e B também, ambos podem ser “sistemas integradores” (função semelhante). Poderíamos ter uma ontologia simples de papéis (e.g., Sistema vs SubSistema vs Mecanismo; Sensor vs Atuador vs Controlador – se marcado) e checar correspondência nessa ontologia de meta-níveis.
12. *Exemplo:* Apoptose vs TTL: ambos têm tag `failure_mode:graceful` → forte correspondência funcional (ambos são mecanismos de término controlado). Sistema Imune vs Observabilidade: se Observabilidade fosse marcada como S1 (operação) e Sistema Imune S1, há alinhamento de papel de defesa operacional.
13. *Explicação:* Score alto indica “exercem funções equivalentes nos respectivos sistemas”. Ex.: “Ambos atuam como mecanismo de isolamento de falha (tag match), ou ambos são controladores de um loop”.
14. **Temporal/Escala & Modo de Falha:** Compara se operam em escalas de tempo semelhantes ou lidam com falhas de maneira comparável. (Esta dimensão complementa a funcional, destacando análogos se ocorrem em frequências ou durações parecidas, ou tipos de falha similares).
15. *Como calcular:* Usa tags específicas como `timescale:` (e.g., realtime, seconds, hours, days, years) e `failure_mode:` (graceful, abrupt, recoverable, etc.). Define-se uma métrica de distância nessas categorias. Ex.: timescales iguais → 1, diferindo um nível (segundos vs minutos) → 0.5, muito diferentes (ms vs dias) → 0.0. Similar para tipo de falha. Faz média das subaspectos ou o mínimo se considerarmos que para temporal+falha estar alinhado, ambos aspectos importam.
16. *Exemplo:* Feedback Negativo em organismos pode atuar em escala de minutos-horas; Backpressure em software atua em milissegundos-segundos – timescales bem diferentes → reduzir analogia apesar de funcionalmente ambos serem controle negativo. Já Apoptose (processo horas/dias) vs TTL (minutos/horas para instâncias) – escalas um pouco diferentes mas não totalmente dispareas; `failure_mode` ambas “programada”, então moderado.
17. *Explicação:* Indica “compatibilidade de contexto dinâmico”. Uma analogia poderia ser conceitualmente boa mas se opera em ordens de magnitude de tempo diferentes, pode ser menos útil – essa dimensão garante a ferramenta sinalizar isso.

(*Opcional - Generatividade/Insight*): Essa dimensão poderia surgir no futuro: mede se juntar A com B gera alguma inferência nova (ex.: uma combinação de conceitos leva a sugerir um terceiro conceito). Porém, isso é exploratório e provavelmente para V3, então não será usado no score MVP. Deixamos mencionado como possibilidade de expandir a explicabilidade (“sugestão de conceito C análogo surge de A-B”), mas não compõe o score nos primeiros releases.

Score Composto (Hotness scalar): A pontuação final entre dois nós (ou regiões do grafo) é uma combinação ponderada dessas dimensões. No MVP, usar pesos fixos decididos por heurística e validação com especialistas: por exemplo:

$$\text{Hotness}(A, B) = 0.4 * \text{Sem}(A, B) + 0.3 * \text{Str}(A, B) + 0.2 * \text{Func}(A, B) + 0.1 * \text{Temp}(A, B)$$

Esses pesos refletem uma suposição inicial de que similaridade semântica e estrutural são os drivers primários, com funcional e temporal refinando. Porém, **exporemos esses pesos para ajuste** (via UI avançada e possivelmente calibrados com feedback). Podemos usar também uma combinação não linear – ex.: exigir algum mínimo em estrutural para liberar uma analogia forte, evitando casos “tudo semântico”. Mas inicialmente, soma ponderada ou média ponderada deve bastar.

Cálculo por camada (Hot_d e agregação): Em grafos hierárquicos, definiremos o score de analogia em cada profundidade. Por exemplo, Hot_0 compara nós de layer 0 (topo), Hot_1 compara layer 1, e assim por diante. O heatmap principal ($A \times B$) por padrão mostra Hot_0 de cada par top-level. Ao clicar num par (A_0, B_0), entramos no sub-heatmap calculado pelos Hot_1 de filhos correspondentes. Poderíamos precisar agregar pontuações de subníveis para refletir no nível pai. Uma estratégia: **propagação com decaimento** – se muitos filhos de A e B se alinham bem, o par pai A-B recebe um boost, mas decaindo com a profundidade. Por exemplo, $\text{Hot}_{\text{final}}(A,B) = \text{peso0} \text{Hot}_0(A,B) + 0.8 \text{peso1} \text{avg_matching}(\text{Hot}_1 \text{filhos}) + 0.65 \text{peso2} \text{avg}(\text{Hot}_2) + \dots$ (*decaindo factor 0.8, 0.65 arbitrários*). **MVP: podemos simplificar e focar no matching de mesmo layer somente***, apresentando as camadas separadamente, sem ainda mesclar. Ainda assim, para decisão de destaque global, poderíamos elevar um par de topo se seus filhos tiveram scores muito altos. Isso requer experimentação – definiremos critérios de “herança de hotness” (ex.: se ≥ 2 pares de layer1 debaixo de A-B > 0.8, então garantir A-B > 0.5 mesmo que nome A-B não fosse obviamente similar). Essa lógica ficará no algoritmo V2+, mas é importante projetar a estrutura para suportar.

Explicabilidade integrada ao cálculo: Cada dimensão fornece *artefatos explicativos*. Portanto, além do número, o motor retorna um objeto contendo: - correspondências de nós ou termos para Semântico (ex.: “ambos contém ‘falha’ no nome” ou “distância de embedding=0.2”). - correspondências de relações para Estrutural (ex.: “A tem X part_of, B tem Y part_of, matched 2 via analogias $X_1 \leftrightarrow Y_1, X_2 \leftrightarrow Y_2$ ”). - tags coincidentes para Funcional (ex.: “ambos marcados como S1 e failure_mode:graceful”). - diferenças notáveis (as divergências): qualquer aspecto onde A e B diferem significativamente, especialmente se penalizou o score. Ex.: sem tags em comum, escalas de tempo diferentes, etc.

No painel “Explain”, exibiremos essas informações de forma legível. Exemplo de apresentação:

- **Top analogias nó↔nó:** Apoptose (bio) ↔ Restart/TTL (sw) – *Semântico*: baixo (0.2, nomes distintos), *Estrutural*: alto (0.9, ambos ocorrem em contexto de controle de falha), *Funcional*: alto (0.8, ambos graceful shutdown), *Temporal*: moderado (0.5, horas vs minutos). -> Score total ~0.7.
- **Relações suporte:** (bio) Apoptose “part_of Homeostase” & (sw) Restart “part_of Autoscaling” (isto contribui, pois Homeostase ~ Autoscaling já mapeados); (bio) Apoptose “regulado por Sinal X” & (sw) TTL

"configurado por métrica Y" – padrão de loop sensor/atuador similar.

- **Divergências:** Apoptose ocorre a nível celular (microsegundo relativo, mas desencadeia cascatas lentas) vs TTL opera a nível de processo computacional (mais rápido); Apoptose irreversível biológica vs TTL reverte quando serviço reinicia (diferente nuance de permanência).

Incluímos também **incerteza/confiança**: por exemplo, se as fontes de dados eram escassas para uma dimensão, poderíamos fornecer intervalos ou marcadores de baixa confiança. Ex.: se não há embedding e comparador lexical encontrou pouca coisa, Semântico=0.5 ± 0.2 (incerto). Ou se analogia estrutural teve que se basear em poucos vizinhos, sinalizar. Esses detalhes podem ser mostrados via tooltip "confiança baixa".

"Hot alto mas risco alto": implementaremos detecção de casos possivelmente falhos: tipicamente quando só uma dimensão dominou. Ex.: Semântico=1.0 (nomes idênticos), demais dimensões = 0 → Score ainda pode ser médio se pesos permitirem, mas claramente é superficial. Sinalizamos tal caso com bandeira de risco (como na história do usuário 10). A lógica: calcule desvio entre dimensões; se uma dimensão >0.8 e todas outras <0.3, ou coeficiente de variação alto, marque "HighRiskAnalog". Outros sinais: analogia altamente dependente de um único vizinho match.

Anti-Goodhart - mitigação no motor: Para evitar otimizações espúrias no score, incorporamos algumas salvaguardas:

- **Cap nos contributos individuais:** Nenhum par deve atingir score máximo só por uma dimensão. Por exemplo, limitar contribuição semântica a 0.7 do total – assim mesmo nomes idênticos não garantem >0.7 se estrutura e função não corroborarem.
- **Diversidade de evidências:** Podemos exigir pelo menos 2 dimensões não-nulas para marcar como analogia "quente". Se só uma dimensão acende, talvez não destacar tanto visualmente.
- **Modo expert:** Fornecer visualização dos componentes do score (barras ou %), de forma que o usuário (especialista) possa ver se o score está "torto". Isso não impede diretamente Goodhart, mas aumenta consciência.
- **Feedback loop humano:** Como definido, usuário feedback treina o sistema; se ele marca uma analogia como "ruim" e nota-se que era porque score semântico dominou, o sistema pode reduzir peso semântico automaticamente um pouco. Em essência, o sistema se adapta para refletir o verdadeiro objetivo (boas analogias reais), não apenas o proxy (score numérico puro).

Em futuras versões, poderíamos até simular "adversários" internos – monitorar se pequenas modificações nos grafos (ex.: renomear conceitos com prefixo similar) aumentam indevidamente o score. Essas heurísticas seriam logadas ou alertadas ("otimização suspeita detectada").

Resumindo, o Hotness Score não é uma mera métrica estática; é um **framework extensível** que combina múltiplos aspectos de analogia de maneira ponderada e explicável, permitindo calibragem e confiança. Ele roda incrementalmente camada por camada e fornece **outputs detalhados** (score + explicações) que alimentam tanto a UI (visualização, tooltips) quanto os mecanismos de alerta e feedback.

E) Algoritmos de Correspondência (Matching) – MVP → V2 → V3

Definimos três níveis de sofisticação para os algoritmos de correspondência de grafos, alinhados com evolução do produto:

MVP (Versão Inicial) – Algoritmo Gulos + Assinatura de Vizinhança:

- *Abordagem:* Para cada nó em Grafo A, calculamos um conjunto de candidatos em Grafo B com base em similaridade local (ex.: top 3 por alguma métrica simples). Em seguida, fazemos correspondência nó a nó de forma gulosa: escolhemos o par com maior score global disponível, marcamos como correspondido, então iteramos refinando – evitando reutilizar nós já pareados muitas vezes. Essencialmente, não garante melhor pareamento global, mas é simples e rápido.
- *Detalhes:* A “assinatura de vizinhança” refere-se a um vetor ou hash que representa o contexto de um nó: ex.: [grau_out(part_of), grau_in(is_a), tem_feedback?, tags...] e podemos usá-la para filtrar candidatos. Por exemplo, um nó de A com assinatura (tem 2 subcomponentes, tag=graceful) só busca em B nós que também tenham pelo menos 1 subcomponente e tag=graceful. Isso reduz busca e impõe *constraints por tipo de aresta* (ex.: se A é top-level sem pais, preferir B top-level sem pais).
- *Custo Computacional:* $O(n * m)$ para comparar todos pares na pior hipótese ($n = |A|$, $m = |B|$). Com heurísticas de filtragem, reduzimos isso. Ex.: se domínios/tags não batem, nem calcula. E caches: pré-computa embeddings e assinaturas uma vez. Para grafos pequenos (<200 nós cada), isso é trivial; para médios (~10k nós), pode ser pesado (100 milhões comparações). Então MVP não pretende suportar $10k \times 10k$ diretamente – confiamos que para esses casos o usuário vai usar filtros ou drill downs.
- *Cache:* Podemos armazenar as similaridades calculadas numa matriz ($A \times B$). Como MVP é offline, memória pode ser um limitante: $10000 \times 10000 = 100e6$ pares, talvez ~400 MB se float – pesado mas não impossível em desktop com 16GB, porém não desejável. Melhor cachear sob demanda: calculado um par, guarda se for acima de certo threshold. O CLI/API pode permitir salvar a matriz ou top-K.
- *Qualidade esperada:* O guloso tende a acertar as correspondências muito óbvias, mas pode errar se dois nós competem pelo mesmo par ótimo. Ex.: nó A1 e A2 ambos similares a B1, o guloso pegará um e o outro ficará subótimo. A qualidade é aceitável para visualização inicial, já que o usuário verá que talvez A2 e B1 também estavam quentes e pode perceber a sobreposição. MVP assume o usuário faz interpretações, não precisando match perfeito.
- *Medição de regressão:* Teremos pequenos benchmarks (grafos conhecidos com “gabarito” de analogias). Mediremos se scores das correspondências chaves se mantêm ou melhoram com alterações de algoritmo. Para MVP, baseline é talvez manual: verificar se quando grafos são idênticos ($A = B$), o algoritmo corresponde cada nó a si mesmo com score 1 nos diagonais. Ou se grafos invertidos (hierarquia diferente) não causa meltdown.

V2 – Correspondência Ótima (Assignment) com Penalidades Relacionais:

- *Abordagem:* Modelamos o matching como um problema de **assignment ótimo** bipartido: cada nó de A pode mapear a no máximo um nó de B e vice-versa (ou poderíamos permitir $n:1$ parcialmente, mas idealmente 1:1 para estrutura principal). A matriz de similaridade (score) é o custo ou peso. Podemos usar o algoritmo Húngaro para achar o matching de custo máximo (ou mínimo de dissimilaridade). Isso garante a seleção global ótima de correspondências, em vez de guloso local.
- Para reduzir complexidade, não faremos com todos pares se for grande – usamos *top-k candidates* para cada nó, e ignore pairs beyond that (sparse matrix). Ex.: para cada nó A, consideramos somente 5 melhores B (com base em heurística rápida). Então resolvemos assignment nesse grafo bipartite esparso. Isso diminui drasticamente o input do Hungarian.
- *Penalidades por inconsistência:* Além dos scores individuais, podemos introduzir penalidades extras se um conjunto de matches viola estrutura. Ex.: se A1->B1 e A2->B2 são chosen, mas há uma relação A1->A2 no grafo A e B1->B2 não existe (ou vice-versa), essa inconsistência estrutural deveria tornar essa combinação menos desejável. Podemos incorporar isso via ILP (programação linear inteira) adicionando variáveis e restrições, ou via um pós-processamento: calcular penalidades para a solução e rejig with heuristics. Full ILP might be heavy, mas para V2 podemos simplificar: primeiro solve Hungarian purely by base scores (que já

incluem algo de estrutura local, esperemos). Depois, detectar maiores violações e ajustar manual: ex., forçar match reconsideration. Ou peso adaptado: antes de solver, diminuir peso de um par se ele cause mismatch com um par já altamente provável.

- *Custo*: Hungarian é $O(n^3)$. Para $n=1000$, $n^3 = 1e9$ ops, impraticável diretamente em Python puro. Mas com top-k ($k=5$), effectively ~5000 nodes considered edges ~5000, Hungarian cost is trivial. So viability depends on how sparse we can make it. We might integrate something like a custom ILP solver (OR-Tools) if needed for advanced constraints. That might be okay for n up to 1000. If extremely large (10k+), maybe skip ILP and degrade gracefully to greedy or cluster-based matching.

- *Cache*: At V2, caching partially matched substructures becomes interesting: maybe store match results for subgraphs to reuse if overall graphs only slightly changed. Also caching top-k for each node (so you don't recompute all to find top-k each time).

- *Qualidade*: We expect V2 to resolve the problems of V1: no more obvious cross-match confusion. The analogias overall should become more consistent: if A1->B1 and A2->B2 because A1 relates to A2 and B1 relates to B2, etc. It might even bring out analogies that were suppressed in V1 because a greedy took a conflicting match. We will measure quality by improvements on benchmarks: e.g., number of correct analogies recovered vs known mapping, and subjective coherence rated by users (like an expert says the analogies shown make more sense as a whole).

- *Regressão tests*: have fixed test sets where known correct mapping yields high total weight; ensure algorithm picks them. Also track performance times – ensure it doesn't blow up unexpectedly for typical sizes.

V3 – Embeddings Estruturais + Re-rankeamento + Aprendizado humano-in-loop:

- *Abordagem*: Nesta etapa incorporamos técnicas de aprendizado para melhorar matching além de regras manuais:

- **Embeddings Estruturais (Graph Neural Networks ou metric learning)**: Treinar ou utilizar modelos para obter vetores que representam um nó *com seu contexto estrutural*. Por exemplo, usar GraphSAGE, GCN ou outro que incorpora vizinhos e atributos (tags) para gerar embeddings comparáveis entre grafos. Poderíamos até treinar no fly a pequeno porte, mas mais realista é usar modelos pré-treinados em ontologias gerais se houver (não trivial, possivelmente usar word embeddings + graph context in a hybrid way). Em V3, o usuário poderia optar por baixar um modelo local (um pacote) que ajuda a refinar analogias.

- **Reranking por constraints**: Mesmo com um modelo sofisticado, mantemos as constraints de coerência. V3 pipeline: gerar lista de potenciais matches ordenados por um modelo (embedding + baseline score), depois aplicar filtros e regras (como no V2) para garantir consistência. Ou inversamente: primeiro usar V2 to narrow consistent matches, then use a learned model to rerank ties or suggest cross-layer analogies.

- **Feedback humano fechado no loop**: Em V3, idealmente incorporamos aprendizado ativo: as marcações “boa/ruim analogia” do usuário alimentam um refinamento de modelo. Talvez um modelo de classificação ou ajuste de pesos via perceptron: features = vetor de dimensões do score + meta (layer difference, etc.), e target = user labeled good/bad. Com poucas labels, talvez melhor ajustar pesos lineares ou thresholds do score (so a simpler approach). Em longo prazo, se acumular muitos exemplos, um pequeno ML model could generalize (e.g., user consistently marks analogies relying only on name as bad – the system learns to deprioritize name-only analogies overall).

- *Custo*: Graph embeddings might be computed offline or on initialization – complexity depends on method, GNN might be $O(V+E)$ per iteration, which for thousands of nodes is fine if done once. The matching step then may become a simple nearest-neighbor search in embedding space (very fast if dimension is modest and using ANN libraries, or just brute force with vector operations due to hardware acceleration). The re-ranking with constraints still maybe ILP or a greedy improvement algorithm – maybe use approximate ILP given we have a good initial solution from embedding similarity.

- *Cache*: We would cache learned embeddings per node, and update them only if graph changes significantly (embedding for new node or neighbors changed – possibly do incremental GNN updates). We also cache the learned model weights from feedback.
- *Qualidade esperada*: V3 deve trazer as correspondências mais “inteligentes”, catching analogies que V1/V2 perdiam porque dependiam de padrões difusos. Por ex., analogias *indiretas* ou *metafóricas* podem surgir – se o modelo aprendeu que “sistema imune” and “circuit breaker” both serve a protective role even se estruturalmente diferentes, ele pode alinhar. Com re-ranking e user feedback, esperamos alta precisão nos top suggestions e menos falso positivos.
- *Medição*: Além de benchmarks predefinidos, V3 seria avaliado por usuários (satisfação com resultados). Também monitoramos se user feedback truly improves future suggestions (i.e., measure some metric of agreement pre/post training).

Resumo dos 3 níveis:

- **MVP**: Regras simples e rápidas, rely on human to interpret, baseline performance.
- **V2**: Introduz rigor matemático (optimal assignment) e considera estrutura global, melhora coerência; computationally heavier but manageable via pruning.
- **V3**: Incorpora machine learning and adaptivity, aiming for deeper analogies and continuous improvement, ensuring it remains local (model training/inference on device).

Benchmarks de regressão:

Teremos, por exemplo, um conjunto de analogias esperadas (maybe handcrafted small graphs or known correspondences like “cat brain vs CPU” etc). For each algorithm version, run and produce top matches, measure precision/recall on that set. Also measure runtime on increasing sizes (100, 500, 1000 nodes) to ensure performance scales as predicted. This will flag regressions – e.g., if a code change makes Hungarian blow up time or a model slows down computation beyond budget.

Em paralelo, mediremos **satisfação qualitativa** (via pilot users) version a version, e rastrearemos incidência de problemas como mismatch ou obviously wrong analogies not flagged – expecting these to drop by V3.

F) Importadores Plugáveis (Input Variado)

A arquitetura de importação é **plugin-based**, permitindo adicionar ou atualizar módulos de leitura sem afetar o core. Cada importador transforma uma fonte de dados específica em um grafo GraphSpec válido.

Contratos de Plugin:

- Cada plugin é uma função ou classe implementando uma interface, por ex.: `def import_data(source_path_or_obj) -> GraphSpec`. Pode registrar metadados como nome, suporte de extensão ou protocolo. Ex.: `import_json_graph` aceita arquivos `.json` compatíveis já quase prontos; `import_text` pode aceitar `.txt` ou `.md`; `import_ontology` para `.owl` / `.rdf`; `import_repo_python` para um diretório de projeto Python.
- O plugin deve retornar um objeto (ou dict) seguindo o esquema GraphSpec. O core então valida e incorpora na base local.
- O plugin não deve persistir dados por si – entrega ao core e este decide salvar. Qualquer cache ou download (no caso de modelo local) deve respeitar restrições (ex.: perguntar antes de baixar).

Lista de plugins planejados:

- 1. import_json_graph:**
2. *Descrição:* Importa um grafo já em formato JSON (talvez exportado de outra ferramenta ou escrito manualmente). Espera-se que o JSON esteja já no esquema GraphSpec ou algo muito próximo.
3. *Entrada:* Path de arquivo `.json` ou objeto JSON em memória.
4. *Processo:* Valida campos obrigatórios (nós e arestas). Se vier de ontologias ou outras ferramentas com campos diferentes, faz mapping (ex.: `nodes` vs `Vertices`). O plugin pode ser trivial se formato já é GraphSpec: só carrega e retorna. Caso contrário, transforma campos conhecidos.
5. *Saída:* Dict GraphSpec.
6. *Validações extras:* Garante `id` único; se não houver `id`, gera a partir de label. Atribui `layer=0` se não fornecido, ou tenta inferir de hierarquia se há pista (ex.: se veem arestas `parent` ou similar). Loga quaisquer assumpções ou correções (ex.: "No layers in input, set all to 0").
- 7. import_text:**
8. *Descrição:* Lê texto livre (artigo, documentação) e tenta extrair um grafo de conceitos e relações. MVP será básico: extrair apenas nós principais e algumas relações simples, mas servirá para experimentos.
9. *Entrada:* Arquivo `.txt`, `.md` ou string de texto.
10. *Processo MVP:* Talvez procurar por padrões como títulos ou listas para definir nós. Ex.: cada linha de bullet ou cada título de seção vira um nó (label = texto resumido). SubPontos viram subconceitos (layer deeper). Ou usar separadores (parágrafos). Para relações, MVP pode não extrair explicitamente, ou usar heurísticas: se texto diz "X é tipo de Y" → aresta `is_a(X,Y)`. Use um mínimo de NLP: talvez spaCy small model (se permitido offline) para extrair sujeito-objeto e verbo entre conceitos. MVP se limita a key verbs ("is", "parte", "causa", etc.).
11. *Saída:* GraphSpec com nós (label = termos encontrados) e edges (type deduzido por frase).
12. *Validação:* Provavelmente incompleta – plugin textual deve marcar grafo com baixa `confidence` meta, indicando que é esboço.
13. *Melhorias futuras:* V2 pode integrar NER (Named Entity Recognition) para achar entidades e relations, ou permitir ao usuário marcar texto (highlight e indicar relação manual assistida). V3 poderia usar modelo de resumo ou ontologia para gerar grafo melhor.
- 14. import_ontology (OWL/RDF):**
15. *Descrição:* Importa ontologias OWL, RDF, RDFS. Converte classes e propriedades em nós/arestas do GraphSpec.
16. *Entrada:* Path `.owl` ou `.rdf` (XML/Turtle etc.), ou até uma URL local.
17. *Processo:* Usar uma biblioteca (ex.: `rdflib` em Python) para parsear. Identificar classes (`owl:Class`, `rdfs:Class`) → criar nós (label = class name, id = URI last segment or assigned). `SubClassOf` → aresta `is_a`. Object properties (`owl:ObjectProperty`) podem ser criadas como edge types or maybe nodes? Melhor: cada assertion "A prop B C" (A C B) vira aresta do tipo = prop from A to B or vice versa depending on property orientation. É um mapeamento que talvez resulte em muitos nós (incl. properties as nodes?), mas para analogias nos interessam principalmente classes as nodes e

property instances as edges. Data properties e individuals provavelmente ignorados no MVP (ou individuals poderiam virar nós also if needed).

18. *Adaptação*: OWL tem muitos detalhes (restrictions, equivalences). MVP: map direto hierarchy e properties. V2: talvez processar ontologia de alinhamento se dois ontos forem comparados (ex.: use owl:equivalentClass as hint de analogia = 1.0 score for those pairs!).

19. *Validação*: Checar duplicatas (mesmo label from different URI spaces?), handle blank nodes if any.

20. *Exemplo*: Uma ontologia simples de gerenciamento poderia ter Classe "Sistema" com subclasse "Departamento", etc. import_ontology transformaria isso para nós "Sistema (layer0)", "Departamento (layer1)", aresta is_a(Departamento, Sistema). Propriedades tipo "temResponsabilidade" possivelmente virar edges `has_role` entre classes ou se instanciadas.

21. *Observação*: OWL é **padrão consolidado** para representar ontologias ⁴, então suportá-lo permite ampla integração.

22. `import_repo_python`:

23. *Descrição*: Varre um repositório de código Python para extrair estrutura de módulos e dependências, formando um grafo. (Poderemos ter `import_repo` para outras linguagens futuramente, mas Python é prioritária).

24. *Entrada*: Diretório raiz de projeto Python.

25. *Processo*:

- Percorre arquivos `.py`. Usa AST (Abstract Syntax Tree) do Python para cada arquivo.
- Identifica top-level classes, funções, e possivelmente constantes importantes como nós. O label pode ser `module.submodule.ClassName` para exclusividade ou separado hierarquicamente: cada módulo/pacote vira nó layer 0 ou 1, classes layer abaixo, funções dentro de classes etc. Precisaremos decidir: possivelmente, *módulos/pacotes como layer0 nodes*, classes dentro módulos layer1, métodos layer2, etc., ou se queremos grafo mais conceitual (depende do uso). MVP talvez considere módulo como nó e classes como sub-nós (layer1).
- Extrai relações:
 - Importações: se módulo A faz `import B` -> edge `depends_on(A,B)`.
 - Chamadas/Usos: se função X chama Y (in another module) -> edge `calls(X,Y)`. Talvez colapsar to module-level for simplicity.
 - Hierarquia: classe extends other -> edge `is_a(Class, BaseClass)`.
 - Contém: uma função pertence a um módulo -> `part_of(Function, Module)`.
 - Armazena também talvez complexity metrics: e.g. number of lines in function, or a tag "@boundary" if function name contains "api" (just example heuristics for tagging). If a directory structure suggests layers (like `services/`, `core/`), maybe tag or layer accordingly.
- Adiciona tags:
 - Classes that end with "Service" or in package "services" -> tag "domain/service".
 - Functions with certain decorators (like `@api_route`) -> tag "entrypoint".
 - Could detect `if __name__=="__main__"` -> mark main script, etc. These tags provide domain context for analogies (like entrypoint ~ sensor perhaps, etc.).
 - (Opcional) **Ω e sinais de "necrose"**: "necrose" sugestiona código morto ou deterioração. Podemos incluir métricas: e.g., no. of TODO comments, last modified date if from git (if old => "stale"), low test coverage if known. Represent "Omega" (Ω) possibly as an overall system

viability metric in VSM context. For MVP likely skip or just tag modules that seem unmaintained as "risk:necrosis".

26. *Saída:* GraphSpec: nodes for modules/classes (with tags like "component", "boundary" etc.), edges of dependencies and composition.
27. *Validação:* Principal é garantir que não gera grafo colossal: se project has thousands of functions, maybe we only include up to certain depth or allow user to choose focus (like perhaps only module-level in MVP). Could default to module level and treat classes as attributes? For richer analysis V2 can refine. Logging should tell summary: "100 modules, 500 classes found; included only modules (set detail=high to include classes)".
28. *Exemplo:* Ao rodar em um repo de microserviços, esperaríamos nós como "ServiceA (module)", "Class CircuitBreaker" etc, edges "ServiceA depends_on ServiceB", "CircuitBreaker part_of ServiceA", etc. Os tags poderiam marcar "CircuitBreaker pattern, failure_mode:graceful". Algumas detecções talvez precisem dicionário de padrões (ex.: classes com nome de padrões conhecidos).
29. *Riscos/Limites:*
 - Repos grandes (ex.: >1000 arquivos) – devemos ter opções de filtragem (ex.: `--max-nodes 500` ou importar somente certos subdirs).
 - Python AST can parse dynamic usage limitedly, but we stick to static structure.
 - Multi-language repos – focus only on Python files for this plugin.
 - We must ensure no code execution, only static analysis (for safety).
 - Variation in code style – might miss relations (like import alias). We'll log "warnings" if something couldn't be resolved (ex.: "dynamic import not resolved").

Plugin extensibility: Document how to add new plugin: e.g., a Java importer could be added similarly (parsing JAR or source via javalang or similar), or a CSV importer that maps spreadsheets to nodes. The core recognizes a plugin by either file extension association or user selection.

Validação do GraphSpec: Após qualquer plugin output, run a validation routine: - ensures mandatory fields present (id, label for each node; source, target, type for edges), - fixes minor things (strip whitespace in labels, ensure layer numeric etc.), - assign default values where missing (e.g., if no domain, set domain=graph.domain if present). - If errors: plugin can throw exception or return error, which UI should show. For partial issues, log and attempt to continue.

All import actions generate a log (in a log window or file) summarizing what happened (ex.: "import_text: 20 candidate concepts found, 15 used after filtering duplicates; relations: 5 is_a, 2 depends_on extracted.").

Logs de importação (debugging): Importers should be verbose in debug mode. E.g., `import_repo_python` might list modules discovered and ones skipped. This aids users in refining input or reporting issues. We might provide a toggle "verbose import" in UI or only via CLI.

Limitar explosão combinatória: If an input source can produce extremely large graphs, importers should apply some heuristics to limit. For example, - import_text: if text is huge (book), maybe only take top N frequent nouns as nodes. - import_repo: if > 1000 functions, maybe collapse by module or only include classes. Or allow user to specify the granularity (like import at module-level only). - import_ontology: user could limit depth of subclass to import. Possibly provide an interactive preview (e.g., list top-level classes and allow selecting subset to import).

In MVP, probably simpler: implement defaults that avoid going beyond medium scale, and clearly document this to user ("import truncated at 500 nodes – refine input or adjust limits"). Perhaps store a flag in meta `truncated: true` if not full, so user knows.

By designing importers carefully, we ensure input variety doesn't break the core – everything becomes a GraphSpec graph of manageable size and consistent structure, enabling the analogy engine to work uniformly.

G) Arquitetura Local-First (Componentes e Storage)

A arquitetura do sistema enfatiza **local-first** em todos os aspectos: o backend roda localmente, dados armazenados local, e a comunicação entre frontend e backend não sai do dispositivo. Vamos detalhar componentes e interações:

Componentes Principais:

- **Interface do Usuário (Frontend):** Aplicação web local (HTML/JS/CSS) rodando no navegador do usuário, servida a partir de um servidor local. Alternativamente, empacotada como desktop app via Electron/Tauri contendo um webview. A UI lida com visualização (heatmap, painéis) e input do usuário (seleções, cliques, feedback).
- **Backend Local (Servidor da Aplicação):** Implementado possivelmente em Python (dado o ecossistema escolhido). Pode usar FastAPI/Flask ou similar para expor endpoints HTTP RESTful na máquina local (ex.: `http://localhost:8000`). Esse backend realiza as operações pesadas: importação, cálculo de analogias, gerenciamento de dados. Ele também orquestra caching e persistência.
- **CLI Interface:** Em muitos local-first apps, a funcionalidade do backend pode ser acessada via linha de comando para automação. Podemos embutir isso no mesmo programa (ex.: invocar unicorn/FastAPI in CLI mode for server, or separate commands). CLI commands: e.g., `analogies import mygraph.json`, `analogies list graphs`, `analogies compute graphA graphB`. Isso aproveita os mesmos componentes do backend mas sem UI.

Fluxo de dados & Endpoints API:

Imagine o usuário abre a UI web. O frontend faz requisições ao backend para ações. Principais endpoints (com exemplo de uso):

- `GET /graphs` – lista grafos disponíveis (nome, id, meta). O backend lê do seu armazenamento (DB ou directory) e retorna JSON.
- `POST /graphs` – importar novo grafo. Body pode conter um arquivo (upload) ou instrução para plugin (e.g., `{"action": "import", "type": "repo_python", "path": "/user/proj"}`). O backend chamará o plugin adequado, obterá GraphSpec, salva local (DB/arquivo) e retorna sucesso ou grafo criado (id).
- `GET /graphs/{id}` – retorna detalhes do grafo (nós, arestas) ou possivelmente apenas meta + content if needed. Para não tratar grafos muito grandes constantemente, poderíamos ter param `?nodes=false` to just get meta.
- `PUT /graphs/{id}` – atualizar grafo (caso de edição manual ou reimport).
- `DELETE /graphs/{id}` – remover grafo.

- `GET /heatmap?graphA=idA&graphB=idB` – solicita cálculo (ou obtenção do cache) do heatmap entre grafos A e B. Pode incluir params como `layer=0` (foco camadas), `dimensions=all` ou filter. O backend verifica se possui resultado em cache (chave possivelmente [idA, verA, idB, verB, paramsHash]). Se cache válido existe, retorna imediatamente os dados (que podem ser: matrix of scores or maybe a partial representation like list of top cells). Caso não, inicia cálculo: carrega grafos do storage, roda o algoritmo (MVP guloso or others), armazena resultado no cache (e possivelmente

meta info like timestamp). Enquanto calcula, poderia stream progress (via WebSockets or via polling on a job id). MVP pode simplificar: block until done if it's quick enough for small graphs.

- Resposta inclui: dimensão da matriz ($n \times m$), e possivelmente uma representação comprimida: talvez não mandar todas células se matrix big? Opcional: retornar só top N analogies sorted, plus a way to fetch full matrix on demand. Mas a UI heatmap precisa possivelmente todos values. We might send a truncated (like values quantized to 256 levels to compress). Possibly just send all as array of floats for MVP (for moderate sizes).
- This endpoint crucially pairs with caching: subsequent calls with same ids and no changes should hit cache.
- `GET /explain?graphA=idA&nodeA=nodeId&graphB=idB&nodeB=nodeId` – retorna explicações para analogia entre nóA e nóB. O backend, tendo previously computed matches, can compile explanation: gather matching context (via stored intermediate calculations). If not cached, might recompute in isolation (which is simpler: just call the scoring function on that pair with extended output). It returns structured data: e.g., `{"semantic":0.8,"struct":0.4,...,"matches": [{"nodeA": "bio2", "nodeB": "sw3", "reason": "both failure_mode:graceful"} , ...], "divergences": ["timescale mismatch"] }`. The frontend will format this.
- `POST /feedback` – body: e.g. `{"graphA": idA, "nodeA": X, "graphB": idB, "nodeB": Y, "feedback": "good"}`. Backend stores this in a local DB table (with maybe fields: graphA, nodeA, graphB, nodeB, label=good/bad/danger). It might also trigger a re-weighting or mark that cache for A-B is stale (if we plan immediate effect). Possibly just store for now and reapply when computing future scores (like loading feedback and adjusting weights or zeroing certain pairs).
- `GET /search?query=abc` – search for nodes matching text across all graphs or a specific graph (we can allow `?graph=id`). The backend uses a full-text index or simple substring search in node labels/tags to find matches. Returns list of hits with context: e.g. `[{graph: id, node: id, label: "...", snippet: "in context"}]`.
- `GET /trend?graph=id&target=somegraph` – this could be for a specialized use-case (v2/v3): retrieving a time series or historical data of analogies (if we track analogies metrics over version history). For MVP, no need; V2+ might support if we store comparisons over time (like how similar is system to baseline vs date). But since listed, possibly a placeholder that fetches stored metrics or differences.

All endpoints operate on local data (no external calls). For larger tasks (like computing a 1000x1000 heatmap), we might spawn a background job (thread or process) so the server can respond quickly and not block other requests. The UI can poll job status or have a websocket event when done. MVP with smaller data might do it synchronously though.

Storage Local:

- **SQLite Database:** Use SQLite for structured data: storing metadata of graphs, index for search, user feedback, and possibly caches indices. We might have tables: `graphs (id, name, domain, version, meta_json)`, `nodes (graph_id, node_id, label, layer, tags_json, embedding BLOB maybe)`, `edges (graph_id, source, target, type, weight, evidence)`. Alternatively, since GraphSpec JSON is our

main storage format, we might just store the JSON as a BLOB or file and only index needed fields in DB (like for search: we could populate an FTS table with all labels and descriptions). So a hybrid approach: GraphSpec stored as JSON file (for easy editing & version control), plus an SQLite for quick queries and cross-graph operations (like search across all).

- **Files:** Directory structure like `graphs/graphname_v1.json`, etc. The GraphSpec is saved. Also store any user exports (images, etc.) by user action, not automatic. If caching heavy results (like a giant matrix), could store in a file (like `.npy` numpy binary) named by a hash, or store in SQLite as separate table or BLOB. Might be easier to store only top analogies in DB for quick access. But at MVP scale, recalculation is fine.

- **Cache invalidation & incremental calc:** Each GraphSpec can have a hash (e.g., MD5) of its content computed on save. When requesting a heatmap, backend compares stored hash with last hash used for the cached result (if any). If any mismatch, mark stale. We keep perhaps a `comparisons` table or directory where a cache file has name combining hashA+hashB+params. If exact match, load it. If underlying graphs changed, we recompute.

- **Incremental recompute:** If one graph changed slightly (like 1 node added), ideally reuse old results for unaffected pairs and just compute new ones. This is advanced: not in MVP, but architecture allows it. We could store partial mapping of node embeddings and similarities, and on small changes, only update relevant rows/cols of matrix. A simpler approach: treat any change as invalidating full analogies at first (MVP), implement more fine-grained later (maybe track subgraph hash per node to see if particular parts changed).

Jobs e detecção de mudanças:

- Implement a mechanism (maybe in backend) to watch the `graphs` directory for changes (if user edits externally) or use explicit signals (UI save). Alternatively, require user to re-import for changes to take effect, which triggers recalc.
- If implemented, an incremental update might: when user edits one node relations, compute a set of affected nodes (that node and neighbors for k hops?), then restrict recalculation to those parts. If performance allows full recompute anyway under 5s, we might not complicate too early.
- We will definitely utilize caching to avoid recompute when user just toggles view or reopens.

Segurança e privacidade:

- *Nenhum dado sai da máquina:* The app does not call external APIs by itself. This includes not loading external fonts or analytics. If it's a web app, host static files from local. If any optional feature requires internet (like downloading a language model), it must be initiated by explicit user action ("Download model X (~200MB)? [Yes]").
- *API Access:* The local API should ideally be restricted to the local machine (binding to 127.0.0.1 or using OS firewall). Possibly require a token for requests to avoid other processes interfering (though if attacker has local access, the risk is low, but still). If multiple user accounts on same machine, binding to localhost is usually fine.
- *Data encryption:* Not strictly needed since all local, but consider if storing sensitive info maybe allow an encryption option or at least obfuscate user secrets if any in logs.
- *Logs:* Default logs should not capture the actual content of user data (like node labels) beyond necessary. For example, log "Graph imported with 20 nodes" rather than listing node names (which could be sensitive). If debugging, user can enable verbose logging at risk of revealing data.

Em suma, a arquitetura consiste em um **backend local leve** (pode rodar persistente ou spawn on demand), um **frontend** rico mas all local assets, e **armazenamento** que combina a simplicidade de arquivos JSON (para interoperabilidade e inspeção) com a eficiência de SQLite (para busca e indexing). Essa arquitetura

garante robustez offline, controlabilidade (todos dados visíveis localmente), e performance (tudo em memória/disco local, sem latências de rede).

(Um diagrama textual simplificado de componentes e interações):

```
[Frontend (Browser UI)]
  ↓ (HTTP REST / WebSocket, Localhost)
[Backend Server (Python FastAPI)]
  ↴ Handlers: /graphs, /heatmap, /explain, /feedback, /search, /...
    ↴ (calls)
      [Importers (Plugins)] -> file system (reads files)
      [Analogy Engine] -> compute scores, uses cache/DB
      [Local Storage Manager] -> read/write JSON files, update SQLite
```

O usuário interage com Frontend; Frontend chama Backend; Backend usa Importers para ingestão, armazena dados via Storage Manager (GraphSpec JSON + SQLite indices). Quando pedida análise, Analogy Engine calcula ou pega cache. Qualquer new data (graphs or results) goes back to storage. Tudo ficando local.

H) Especificação de UI/UX (Layout, Navegação, Estados)

O design da interface foca em permitir exploração fluida dos heatmaps de analogia, com navegação intuitiva entre níveis e acesso transparente à explicação e controle. A seguir, descrevemos a tela principal, interações e estados.

Layout da Tela Principal:

- **Barra Superior:** Contém seleção dos dois grafos a comparar e controles globais.
- **Seletores Graph A / Graph B:** Dois dropdowns ou painéis de seleção, listando os grafos importados (nome e versão). Exibe também um botão “Importar...” para chamar diálogo de import (acionando plugin conforme tipo de arquivo selecionado). Uma vez selecionados Graph A e Graph B, um botão “Comparar” (ou automaticamente) gera o heatmap.
- **Controles de Camada:** Uma opção/slider para escolher qual camada visualizar ou até que profundidade incluir. Ex: “Camada: 0” (top-level only) por default. Se usuário aumenta para 1, pode mostrar analogias agregadas incluindo layer1? (Precisa decidir: possivelmente toggling layers to include). MVP talvez só mostre layer 0 no heatmap principal e drill-down for deeper. Mas podemos ter um controle “Mostrar subníveis combinados” – melhor deixar para V2.
- **Controle k-hop:** se oferecido, define quantos níveis de vizinhos considerar no cálculo estrutural. Ex: “Contexto relacional: 1-hop (padrão)” podendo 2-hop. Isso afeta recomputação, então se mudado, indica recalc needed.
- **Filtro Top-K:** Uma opção para focar nas top-K analogias globais: ex. “Exibir apenas top 100 matches” – util se grafo grande (pode reduzir clutter). Como UI, talvez um slider “Mostrar % top analogias” ou um campo. Se ativado, a matriz mostra somente essas correspondências (talvez destaca pontos e suprime as outras?). Ou highlight mode.
- **Pesos do Score (Avançado):** Um botão “Ajustes Avançados” que expande um painel com sliders para Semântico, Estrutural, Funcional, Temporal. Sum 100% ou similar. Usuário ajusta e clica “Recalcular” (ou auto, se não muito pesado). Por default, esse painel fica escondido.

- Outros botões: “Salvar/Exportar” (para exportar imagem do heatmap, ou relatório JSON), “Configurações” (gerais, como tema claro/escuro, toggles de grid, etc.), e possivelmente um indicador de status (ex.: green dot if up-to-date).

- **Área Central – Heatmap Principal:** O elemento visual principal é uma matriz onde:

- Eixo X = nós do Graph B, Eixo Y = nós do Graph A (ou vice-versa, mas fixar padrão, digamos A vertical, B horizontal). Cada célula representa a analogia entre nó_i de A e nó_j de B, colorida conforme o hotness score. Eixos têm rótulos com nomes dos nós (ou IDs abreviados se muito longo, com tooltip full name on hover).
- **Zoom/Pan:** Se grafos > ~30 nós, a matriz pode não caber. Implementar scrollbars or click-drag to pan across it. Zoom with pinch or ctrl+wheel to see more detail or overview. Possibly an overview mini-map if huge (like a small thumbnail in corner).
- **Tooltip on Hover:** Ao passar o mouse sobre uma célula, mostra um pequeno tooltip com: “A.label ↔ B.label: Score X (e possivelmente mini breakdown: e.g. S:0.8, St:0.5...)”. A coluna e linha labels também highlight (maybe highlight the row and column). This gives quick insight without clicking.
- **Selection:** Clicking uma célula seleciona aquele par. Visualmente, pode contornar a célula (highlight border) e fixar its state (so tooltip stays or highlight stays even if mouse moves). Isso aciona atualização do Painel Direito (“Explain”) com detalhes do par selecionado.
- **Legendas:** Próximo ao heatmap, uma legenda de cores com escala (e.g., azul=0 frio, vermelho=1 quente, passando por cores intermediárias perceptualmente uniformes ). Deve haver indicação do valor mínimo/máximo exibido (as vezes 0-1, mas se normalizamos or threshold top-K, etc.).
- **Largura dos labels:** If many nodes, labels on axes can overlap. Possibly rotate text 90° on top axis for readability, and allow horizontal scroll for left axis if needed. Or show only subset ticks and expand on hover. For MVP, if nodes < 50, just show all; if more, might skip some labels.

- **Painel Direito – “Explain & Actions”:**

- This panel is context-sensitive. By default (no cell selected), poderia mostrar instruções (“Selecione uma célula para ver detalhes da analogia”), ou talvez lista das top analogias global (ranking) as clickable entries.
- When a cell (A_i vs B_j) is selected, it shows:
 - **Resumo do Par:** nomes completos, tal vez com domínio/ícone, e o score total. If they have descriptions, could show a short description each to remind user of meaning.
 - **Detalhamento (explain):** structured list or text:
 - **Matching Features:** e.g. “Tags compartilhadas: failure_mode:graceful, control_loop:feedback_negativo (+0.2)”
 - **Relações alinhadas:** e.g. “Ambos têm subcomponentes análogos: Apoptose ~ TTL, Necrose ~ CrashLoop (contribuiu +0.3)”
 - **Diferenças:** e.g. “Escala temporal distinta (dias vs minutos) (-0.1 no score)”.
 - Possibly show a mini-table of the dimension scores: a small 4-row table or bar chart for semantic/structural/functional/temporal scores to visualize contribution.
 - This section should be scrollable if content long, and maybe collapsible subsections if too detailed.
 - **Divergences emphasized:** maybe highlight in red the divergences.

- **Ações (Feedback):** Below explanation, provide buttons:
 - “ Boa analogia”, “ Ruim”, “ Perigosa” (the last perhaps means “pode levar a engano”). These map to feedback categories. On click, perhaps slight change (like button stays highlighted to indicate recorded). Possibly allow undo (click again to un-highlight).
 - Maybe a “Add note” where user can add a small comment on this pair for their own records (this could be stored but optional).
- **Drill-down Action:** If the selected nodes have subgraphs (layer + 1 exist), an action “ Detalhar Subconceitos” appears. Clicking it triggers loading of a *sub-heatmap view*. Implementation: we can either replace the main heatmap or open a new tab/section. Preferably, we replace the matrix with a focused view of just children of those nodes. The UI might visually indicate context (like breadcrumb above: “Homeostase ↔ Autoscaling” as title of sub-heatmap, with a “← voltar” to main view). Alternatively, could open side-by-side, but that complicates screen estate. A simple approach: clicking drill-down, the main heatmap transitions (maybe animate zoom into that cell) and then shows new matrix. The axes now show A_i's children vs B_j's children. The right panel could now show specifics of that submatrix or remain similar for cell selection there.
 - Provide a “Voltar / Up one level” button to return to previous heatmap context. Possibly above the heatmap or in the breadcrumb.
- **Cross-highlighting:** As user hovers a particular match either in heatmap or in explanation panel, we highlight related elements:
 - Hover on a cell: highlight the corresponding nodes on axes (maybe bold or underline label on those axes). Optionally, highlight any other cells in same row or column that are notably hot, to see how that concept compares to all others. But careful not to clutter. Possibly highlight entire row and column background.
 - Hover on a node label (axis): highlight that row/column and maybe show a tooltip summary of that node (e.g., “5 subs, tags:...”). If user clicks a label, we could consider showing all analogies for that node sorted (like fix that row as context in the side panel). MVP not required, but an idea (e.g., right-click on label = focus node?).
 - In explanation panel, if listing e.g. “Apoptose ~ TTL”, hovering that text could flash or circle those in the heatmap (if visible) or the positions within the sub-heatmap. Or if not visible, at least highlight their labels in the panel if present. Might be complex, but at least we highlight if in main matrix.
- **Feedback loop UI:**
 - After user marks analogia boa/ruim, we may provide subtle acknowledgment: e.g., a tiny ✓ appears or the button stays colored. If enough feedback given, perhaps the system suggests “Recalibrar scores?” or does it silently. Possibly an “Apply feedback” button if we want user to trigger recalibration explicitly (so they can mark a bunch first). For MVP, simply store and maybe adjust weights gradually behind scenes. Or simpler: no immediate recalculation, just store and future computations will incorporate (less confusing perhaps). But user might expect to see effect, so maybe pressing feedback could fade that cell slightly if “ruim” (visually acknowledging it'll be de-emphasized).

Estados da UI:

- **Empty state:** No grafos ainda. UI sugere importar ou criar. Possibly a welcome message and a big "Import Graph" button.
 - **Loading state:** When a heavy computation (import or heatmap calc) is ongoing, show a spinner or progress bar. For import, if possible a progress (like percent of file processed if ontology etc.). For heatmap, we could show progress in terms of pairs done if calculable. If not, a generic spinner "Calculando analogias...". Possibly blur or disable interface parts during calc, but better allow some cancel if taking too long.
 - **Cached state:** If a heatmap result is loaded from cache quickly, perhaps show a small cache icon or "(cache)" note, mainly to convey it might be outdated if graphs changed. Also a button "Recompute" if user wants fresh.
 - **Stale state:** If we know a graph changed (via version/hash), and user has a heatmap from old version, mark it stale. UI can overlay a light red warning "Resultado desatualizado, recalcule para dados atuais." Possibly auto-recalc after short delay or prompt user.
 - **Error state:** If an import fails (e.g., file parse error), show message in a modal or in an "Import Log" area: e.g., "Erro ao importar ontologia: sintaxe inválida na linha 42" – actionable info. If heatmap calc fails (shouldn't normally, but e.g., out-of-memory), catch and show "Não foi possível calcular analogias: memória insuficiente para matriz, tente reduzir top-K ou camada." Provide suggestions. - All errors should be presented in user-friendly language, possibly with a tech detail if hovered (for advanced users).
 - After error, allow the UI to recover (not freeze). Perhaps return to previous state.
-
- **Success/Done state:** After each action, a short non-intrusive confirmation (e.g. import success: "✓ Grafo 'X' importado com 27 nós." maybe as a toast message). On compute done: could highlight some result or just remove the "Calculating..." overlay.
 - **No Data state:** If user selects two graphs but one is empty or they share no comparable nodes, heatmap blank (all zeros?). Should indicate "Nenhuma analogia encontrada ou grafos vazios" if truly nothing.
 - **Interaction states:** Ensure hover vs selected vs disabled states are visually distinct in UI components (buttons, list items). E.g., Graph list entries highlight on hover, selected graph perhaps with checkmark or different background.

Export funcionalidade:

- A menu or button "Exportar" offering:
 - "Exportar Heatmap PNG" – triggers backend or frontend to produce an image (taking into account current zoom or entire matrix?). Possibly backend easier: render via an offscreen canvas or use the same data. Or allow front to do it if it can capture its canvas (modern browsers allow canvas to dataURL, but if using WebGL might need a different approach).
 - "Exportar Heatmap SVG" – potentially just the axes and a vector representation of colored cells (maybe heavy if large). Could be fine for smaller or for the aggregated view. If not trivial, maybe later version.
 - "Exportar Dados JSON/CSV" – could output the matrix of scores or top analogies as JSON or CSV for analysis. Or the GraphSpec of combined info. Possibly an "Analogy Report" JSON containing the pairs above threshold with their breakdowns. This is useful for offline analysis or writing papers.
- When user triggers export, show file dialog or auto download in browser.
- Possibly allow copying an image to clipboard for quick paste.

Estética e usabilidade:

- *Design System*: Use an 8px grid for spacing – e.g., padding in panels multiples of 8, consistent margins. Use typography hierarchy: e.g., axis labels maybe 12px bold sans-serif, panel headings 16px, body text 14px. Ensure high DPI support by using vector/SVG text or proper canvas scaling (discussed in visual section).
- *Color scheme*: Likely a neutral background (white or light gray) for the app, with the heatmap colors providing contrast. Or dark mode optional. The colormap itself should be perceptual (viridis or similar) to avoid misleading interpretation ². Avoid pure red-green for accessibility (viridis is colorblind-friendly).
- *Tooltips*: Appear near cursor after small delay, no flicker (perhaps only show after 200ms hover to avoid spamming when just moving across matrix). Use a stable position or slight offset so the cursor doesn't exit the cell accidentally.
- *Responsive UI*: Although primarily desktop-focused, ensure elements rearrange if window is resized (e.g., if narrow, maybe panel moves below the heatmap). Performance UI likely not used on mobile, but making it somewhat responsive is good.

Navegação geral:

- The UI should allow the user to easily move between tasks: e.g., after import, automatically perhaps select that graph in one of the slots to encourage next step. Or if multiple graphs loaded, maybe a multi-select compare view (maybe not needed if just pairwise though).
- If user wants to compare another pair, they simply change dropdowns and the heatmap updates (maybe automatically if auto-run on select, or requires hitting Compare button if computation heavy to avoid accidental recalcs). Possibly have an “auto compare” toggle.
- *State persistence*: If user imported and compared, then closed app, on reopen we might remember last selected graphs and reload that comparison (as per user story 9). Could store in local config.

In sum, the UI should feel like a polished analysis tool: smooth interactions, clear visuals, and stable behavior even as data changes or gets large. By providing intuitive controls for drilling down and clear feedback on actions (like import or errors), we ensure the user can focus on exploring analogies rather than fighting the interface.

I) Visualização Top-Tier (Renderização e Performance)

A visualização do heatmap e gráficos relacionados precisa ser **impecável**, conciliando qualidade gráfica (nitidez, cores adequadas) e performance (interatividade em tempo real). Aqui detalhamos as estratégias de implementação gráfica:

Tecnologias de Renderização:

- **Canvas 2D vs WebGL vs SVG**: - Para o heatmap principal (matriz possivelmente grande), o **Canvas 2D** é apropriado no MVP devido à sua simplicidade e desempenho razoável até certas dimensões. Podemos manipular diretamente pixels ou desenhar retângulos coloridos. Canvas bem otimizado consegue algumas centenas de milhares de retângulos, mas para milhões de células WebGL se tornaria necessário.
- Para escalas maiores (V2/V3, matrix > ~1000x1000), moveremos para **WebGL**: podemos passar a matrix como textura para um shader e renderizar em um único draw call. WebGL lida com centenas de milhares de pontos facilmente ⁵, garantindo fluidez mesmo em datasets enormes (500k+ pontos) ⁵. Além disso, WebGL facilita **tiling** e zoom sem recravar DOM elements.
- **SVG/HTML**: Será usado para elementos que requerem alta nitidez e interatividade text-based, como labels de eixos, ícones e overlays como highlights ou small graph diagrams in explain. SVG mantém o texto cristalino em qualquer DPI e é fácil de style via CSS. Mas SVG não é bom para milhares de tiny rectangles

(heavy DOM). Então o plan: - Heatmap cells: Canvas or WebGL (raster). - Text and possibly axis lines: DOM or SVG. For instance, each label on the axis can be a rotated `<div>` or `<text>` in SVG. The number of labels = number of nodes, which could be 100s (maybe up to 1000 in extreme), which is borderline but likely fine if handled carefully (1000 text elements is okay). We could also draw text on canvas but then scaling/zooming and DPI issues occur.

- Tooltips: regular HTML overlay.

Pixel Perfection & DPR (Device Pixel Ratio):

- On Canvas, account for DPR to avoid blur: e.g., if canvas intended visible size is 500x500 CSS px and DPR=2, set canvas width=1000 height=1000 and scale down via CSS, and in drawing context scale things accordingly. This yields crisp lines on retina displays.
- In WebGL, similar approach: adjust viewport and possibly use high-res textures or MSAA for line edges if needed.
- All UI icons and text should also scale properly – using vector assets or appropriate font sizes.

Tooltip sem flicker:

- We'll implement tooltips carefully: Possibly using a single tooltip element and moving it rather than destroying/creating for each cell. On `mousemove` over canvas, we can compute which cell (if any) is under cursor (with math given x/y and matrix indices – trivial if fixed cell size and pan offset known). Then only if cell changes, update tooltip content and position. This prevents flicker because rapid moves within same cell won't hide/show repeatedly.
- The tooltip could appear after a small delay to avoid popping up when user is just panning. Perhaps show only if mouse stays for >100ms on same cell. On exit cell, hide after similar delay.
- Use a semi-transparent background for tooltip box and small fade-in effect to look smooth.

Hover "barato":

- We avoid re-render of entire matrix on hover. Instead, highlight effect can be achieved via a second canvas or an overlay effect. For example, highlight row/col: we could draw two highlight rectangles (one across the row, one across col) on an overlay canvas, or simply use CSS to change style of labels on axes (which is minimal). For highlighting the cell itself, perhaps drawing a border rectangle in an overlay or using an HTML element for that cell if we know coordinates.
- The key is to precompute mapping from matrix indices to pixel coordinates. If matrix is large and scaled down, might need to map continuous coordinates if zoomed out. But we can figure relative easily.

Colormap Perceptual:

- Avoid the infamous rainbow (Jet) because it distorts perception³². Use colormaps like **Viridis**, **Inferno**, **Plasma**, or **Cividis** which are perceptually uniform (monotonic luminance)². Viridis, por exemplo, varia suavemente do azul ao verde ao amarelo com brilho monotonicamente crescente² – ótimo para destacar amplitude sem artefatos. Isso significa que diferenças no valor correspondem a diferenças percebidas consistentemente². - Provide maybe 2-3 palette options (all perceptual and colorblind-friendly) in settings: e.g., Viridis (multi-color), Gray scale (if user prints in B/W or is colorblind), and maybe a diverging if we had positive/negative values (not applicable here as all scores 0-1). But default to Viridis or similar. - Implementation: If using Canvas, either precompute a 256-color lookup table from chosen colormap and map scores to that. Or use a shader in WebGL to map value to color via piecewise function or texture.

Acessibilidade (Accessibility):

- **Contraste:** Ensure the colors chosen are distinguishable. Viridis has good contrast even for colorblind ². Additionally, the UI around (text, backgrounds) should meet contrast guidelines (e.g., text on panel backgrounds at least 4.5:1 contrast). - If a user is colorblind severely or prints grayscale, the monotonic luminance ensures they can still decode it somewhat (Viridis goes from dark to light). Alternatively, allow switching to a grayscale or a high-contrast diverging map that is colorblind safe (like blue-orange). - **Alternative representations:** Possibly allow the user to hover and see numeric value, or toggle overlay of contour lines if continuous, but for now, tooltips suffice. - **Keyboard navigation:** We might not implement fully in MVP, but ideally allow focus on elements: e.g., tab to move to heatmap, arrow keys to move selection cell by cell, with some text readout. The complexity might be high to do fully, so in MVP, focus on visual clarity.

Performance Budget & Techniques:

- **60fps pan/zoom:** Achieved by ensuring that panning doesn't trigger a full redraw of all cells in a slow way. Approach: If canvas, drawing entire matrix might be heavy (if 1000x1000, that's 1e6 draws which is too slow per frame). Instead, implement **tiling**: - Pre-render the heatmap in tiles (say 256x256 pixel chunks). For each tile, we have precomputed image (e.g., an offscreen canvas or an ImageBitmap). When panning/zooming, simply blit these tiles into the main canvas at appropriate positions. Only when zoom level changes significantly or we zoom in beyond tile resolution do we re-render (or fetch next LOD). - Alternatively, if WebGL: we can have the entire matrix as a texture and use the GPU to handle zoom/pan (just adjusting texture coords in a quad). That would be extremely fast (essentially the entire matrix is one object). In that case, 60fps is fine even for very large (since it's just moving a textured rectangle). We have to update texture if underlying data changes or if we zoom in beyond resolution – we might use a larger texture or dynamic update. But presumably, a 2k x 2k texture is fine on modern GPUs. If bigger, tiling textures might be needed (but some WebGL allows huge textures, though might hit GPU memory). - **Tempo máximo para render inicial:** We set a soft target: e.g., <1s for initial draw for moderate graphs (<100x100). Possibly up to ~3s for bigger, but ideally not block UI thread. We can show partial loading if needed. If using WebGL, uploading a large texture might also stutter, so consider chunking or showing progressive resolution (like first draw a downsampled version, then refine). Possibly not needed if within a second. - **Hover/tooltip response:** Target ~<100ms from hover event to tooltip shown. That means efficiently computing which cell: e.g., if we have the matrix values in a JS array, and each cell size maybe dynamic if zoomed, calculating index = floor((mouseX - offsetX)/cellWidth) is trivial math. Then look up value from array or precomputed structure. That should be <0.1ms. So the main cost is updating DOM for tooltip, which is fine. - **Memory considerations:** A 2000x2000 matrix of float values is ~4 million floats ~32 MB. Manageable. The rendered canvas at DPR2 might be 4000x4000 pixels = 16 million pixels. If each pixel drawn as one rect in Canvas, not feasible per frame. But if prerendered or drawn via putImageData, possible. Or as WebGL texture 16 million px ~ 64MB (if 4 bytes each), okay on GPU. So definitely need optimization beyond brute force per pixel in JS. - Therefore, techniques: - Offscreen canvas: we can use an OffscreenCanvas in a WebWorker to generate image data array and then transfer the ImageBitmap to main thread to draw. That moves heavy calc off main thread. E.g., create an array of RGBA for each cell or each tile. For moderate sizes, might not be needed, but for bigger, yes. - Use `ImageData` + `putImageData` to batch draw rather than drawing millions of rect calls. If we have the data in a flat array, converting to pixels might be fastest by just mapping values to color bytes and one putImageData. That is possibly the simplest near-C performance approach. - For even bigger, WebGL with fragment shader reading from a 2D texture (or computing color from a formula if we pass score in some buffer) would push heavy lifting to GPU. - **Tiling:** If matrix is extremely large (like 10k x 10k), we cannot load full resolution at once. So concept of LOD: at zoomed out, we render a downsampled (like 1000x1000 or less) version. Only when zoomed in to a quadrant, we load that tile at full

detail. Similar to how map services load higher resolution tiles as you zoom. Implementation: if user zooms, determine visible region and resolution, load appropriate tile images. Could generate tiles on the fly (maybe heavy). Alternatively, require user to explicitly ask for very large analysis, which might not be initial scope. But we plan approach so it's possible in future.

Visual Regression Testing:

- We'll capture reference images of the heatmap and key UI states (maybe using a headless browser) and store them. Each code change can re-generate and compare histograms or pixel diffs. For minor differences (like antialiasing changes), allow small threshold. This ensures we don't inadvertently degrade color mapping or rendering clarity. - E.g., ensure that a known small matrix yields exactly the expected colored output (we can precompute those color values).

Design Guidelines (Look & Feel):

- **Tipografia:** Use a clean font (system sans-serif or something like Inter) that is easily readable at small sizes. Ensure consistent font sizing – maybe base font 14px for UI text, 12px for axis labels (if rotated, consider effective size), 10px minimum anywhere. Titles slightly larger (16-18px). - **Grid & Spacing:** 8px base grid: e.g., 16px padding around panels, 8px margin between buttons, etc. This yields a harmonious layout. - **Hierarchy:** Use visual hierarchy for ease: Panel titles bold, section headings, etc. The heatmap itself is the focal point, so other UI is slightly toned down not to distract (e.g., neutral colors, minimal chrome). - **Tooltips style:** consistent with overall theme, e.g., light background with slight shadow, small font if needed, but ensure readability. Possibly include an arrow pointing to the cell. - **Legendas:** The color legend should have ticks or labels for min/mid/max values. Might put it to the right of heatmap or below, with a gradient bar. Ensure it's labelled e.g. "Analogias Fracas" to "Fortes" or numeric 0.0, 0.5, 1.0. Possibly incorporate it into the panel or as overlay. - **Eixos:** Possibly draw lines or shading to separate sections if needed (if group nodes by domain). But if just listing, maybe zebra stripe backgrounds on labels for readability if long list. But careful not to clutter cell area. - Provide subtle guidelines or dividing lines if matrix large, maybe every 10th cell line slightly thicker, to help track position. Or allow toggling a grid overlay. - If using dark mode, ensure colormap tested on dark (viridis works on both). Possibly invert UI colors accordingly.

The end goal: a user with a high-DPI monitor sees a **razor-sharp** heatmap, with no blurry text or icons (thanks to proper DPR handling), the colors are **scientifically chosen** to convey data accurately ², interactions feel **snappy and intuitive** (no lag when exploring), and the design is aesthetically pleasing (consistent spacing, clear fonts, balanced layout) on par with large tech product interfaces.

J) Plano de Testes (Qualidade e Confiabilidade)

Para garantir que o produto atenda aos requisitos com confiança, definimos um plano abrangente de testes:

Testes Unitários:

Foco em partes isoladas: - **Cálculo de Score:** Funções que computam cada dimensão (semântica, estrutural, etc.) receberão casos simples. Ex.: Semântico: dois textos iguais -> score ~1; totalmente diferentes -> score ~0. Estrutural: dois nós com idênticas assinaturas -> 1; sem interseção -> 0. Assegurar combinações e limites (empty tags, etc.).

- **Combinação de Score:** Dado vetores de dimensões, verificar se o composto sai correto conforme pesos. Testar casos de dominância para ver se anti-goodhart capping funciona (ex.: 1.0 + 0 + 0 -> não ultrapassa

limite definido).

- *Importers*: For each plugin, prepare minimal input and expected GraphSpec. Ex.: `import_json_graph` with a known JSON and verify it returns correct node count and structure. `import_text` - feed a small text like "X is a Y." expecting nodes X, Y, edge X is_a Y. `import_repo_python` - create a tiny dummy repo in temp (a file with a class and import) and ensure we get correct nodes/edges. Use temporary directories in tests to simulate user files.
- *Parsing/Normalization*: e.g., ensure label normalization (IDs generated from label) yields unique and trimmed results. If label conflict, does system append suffix, etc.
- *GraphSpec validation*: feed an invalid graph (duplicate IDs, missing fields) to validation function, ensure it catches and returns error or fixes as expected.
- *Feedback logic*: test that when a feedback entry is added, the data store is updated, and if we simulate a recalculation using that feedback, the weights adjust appropriately (if implemented immediate). Or test that feedback retrieval works.
- *Utility functions*: search query parsing, caching key generation (ensuring stable and unique keys for graph combos), etc.

Each unit test should be small, deterministic, not relying on external state (use test-specific data).

Testes de Integração (End-to-End):

Simulam fluxos completos: - *Import → Heatmap → Explain*: Example integration scenario: Prepare two small graphs (maybe as JSON or via code construct) representing a known analogy. Use CLI or API to import them. Then call the compare endpoint (or function) to compute heatmap. Verify the output contains expected hot cell at the known analogical pair (within some tolerance). Then call explain on that cell and verify content includes the known matching info. Essentially, test that pipeline connects correctly: input transforms -> calculation -> explanation.

- *UI End-to-end (if possible)*: Using a headless browser (like Puppeteer or Selenium) to simulate a user: open the app, import or select graphs (we could pre-load by placing file in correct folder), click compare, wait for heatmap, click a cell, verify that the explanation panel DOM contains expected text. This catches any integration issues between frontend and backend. We should at least do this for a small known scenario.
- *Cross-component*: If an ontology is imported then search is used, verify search finds a known term from that ontology. Or after marking feedback, simulate a recompute to see effect (if implemented immediate effect, else just ensure feedback is stored).

Testes de Desempenho:

We set up automated measurements: - *FPS test*: Use a script with a dummy UI (or instrumentation in code) to simulate panning the heatmap back and forth and measure frame rate. Possibly use browser performance API or custom triggers: e.g., for a moderate matrix (say 100x100), ensure >55fps average. For a heavier one (500x500), ensure at least 30fps. This might be part of an integration test that runs in a headless GPU-enabled environment. If not easily automated, at least manual measurement guidelines.

- *Tempo de render*: Instrument the code to log time from user hitting compare to heatmap fully rendered. Write tests that assert it's below threshold for certain sizes. E.g., generate two graphs of 50 nodes each, measure analogies calc time (should be <1s ideally). For 200x200, maybe <5s, depending on hardware. We may not auto fail if above, but track it as a metric to avoid regressions (maybe if time doubled unexpectedly after a change, flag it).

- *Memória*: We can use memory profiling (perhaps in Python tests for the backend) to see that importing or computing does not leak objects. E.g., run 100 import and remove, ensure memory usage roughly constant. For front, maybe not automated but keep an eye via Chrome devtools in manual tests.

Testes de Regressão Visual:

- Using a consistent environment (same OS, DPI, headless if possible with fixed DPR), render key visual components and compare to baseline images:
 - A known small heatmap (like a 5x5 with known values pattern) – produce an image via export function, compare pixel by pixel to baseline image (with tolerance to account for maybe minor aliasing differences). If we change colormap or drawing code inadvertently, this catches differences.
- UI elements: perhaps take snapshot of an explanation panel example and ensure text is as expected. Possibly verify no overlap or cutoff.
- We might rely more on manual acceptance for UI styling, but a basic heatmap output test is valuable.

Dataset de Benchmark:

- Prepare a *mini dataset* (small graphs with known analogies) and *médio dataset* (e.g., around 100 nodes each, with some expected analogies). The mini can be used in automated tests thoroughly (like every commit, run it). The medium can be used for performance runs and occasional integration tests.
- Possibly include the example concepts (biologia vs software list) as a medium test: we expect certain pairs to be top (Apoptose vs TTL, Homeostase vs Autoscaling, etc.). We can encode expected "top 5 analogies" and verify they appear in top results of the output. This also double-checks correctness of algorithm qualitatively.
- Over time, expand with more benchmarks as we gather them.

Criteria de Aceitação (checklist):

We will derive specific acceptance criteria from each deliverable and user story. For each, define tests or verification steps. For example:

- "Interface render nítido em high-DPI" -> manual test: open on retina screen, verify no blurry elements (especially text). Or screenshot and pixel-detect crisp edges.
- "Heatmap + drilldown + explain funcionam" -> integration test: do drilldown sequence and check that things update accordingly (like ensure after drilldown UI shows new axes labels).
- "Nada sai da máquina" -> perhaps scan the code for network calls, or run app with a tool that monitors network (in test mode intercept fetch, expecting none). Could include a test that tries to do an API call externally (like stub out fetch to see if used).
- "Persistência local" -> automated: run actions, close app (simulate by restarting backend), reopen and ensure previous state is loaded (we can simulate by calling backend after restart to see if graphs still present).

We maintain a **checklist** (like in section M) for final QA, which essentially enumerates key features and properties. We ensure each has at least one test or manual verification tied to it.

Additionally, incorporate tests for edge cases:

- Extremely small graphs (0 or 1 node) -> ensure no crash, UI handles gracefully (we can unit test computing analogies with empty input yields empty or trivial output).
- Non-ASCII characters in labels (accented, symbols) -> ensure they pass through (test e.g. label "São Paulo" remains intact, search can find "Sao" maybe).
- Large tag lists or deeply layered -> ensure no recursion crash.
- Cancel operations: if we had cancel, test that it stops and UI recovers.

Continuous Testing: - Setup these tests to run on each commit or regularly, using a CI environment that can handle perhaps a headless browser for UI tests and has a typical CPU to catch performance (or at least relative changes). - Performance tests might not be strict pass/fail due to environment variation, but can log metrics to track trends.

User Testing & Feedback: While not automated, as part of test plan we include running the tool by some pilot users (domain experts from our persona categories) and gathering feedback on correctness of analogies and ease of use. Use their feedback to tweak parameters and identify any misinterpretation or UI confusion (which then become issues to fix and possibly add tests to ensure improvements stay - e.g., if a user found tooltip flicker annoying, after fix, we ensure via a small stress test that tooltips no longer flicker with certain motion patterns).

By combining rigorous automated tests with user-centric evaluation, we aim to deliver a robust and reliable product with confidence in each update.

K) Roadmap (MVP → v1 → v2 → v3)

Desenvolveremos o produto em fases, cada uma com milestones claras e critérios objetivos de “pronto/done”:

MVP (Minimum Viable Product): Foco em entregar o núcleo funcional básico que já permite a exploração de analogias localmente. - **Importar GraphSpec JSON:** *Meta:* O usuário consegue importar (ou carregar) grafos já estruturados em JSON no formato GraphSpec. Critério de done: Suportar pelo menos arquivos .json que contenham nós/arestas; após import, o grafo aparece na lista com contagem correta de nós. Teste: importar um JSON de exemplo e verificar no UI (ou via API) que o grafo está armazenado e exibível. - **Calcular heatmap por camada 0:** *Meta:* Dado dois grafos importados, o sistema calcula a similaridade de cada par de nós top-level (layer 0) e renderiza um heatmap interativo. Done: Visualização do mapa de calor exibindo diferenças de cor conforme scores, com legenda. Teste: usar grafos simples com 1-2 pares obviamente similares e ver se a célula correspondente está mais “quente” que outras. - **Painel Explain para célula selecionada:** *Meta:* Ao clicar numa célula do heatmap, abre-se painel com explicação textual do porquê da analogia (pelo menos listando tags comuns e possivelmente um relacionamento correspondente). Done: Para pares conhecidos no dataset de teste, o painel lista pelo menos um elemento correto de explicação (ex.: "Tag compartilhada X", ou "Ambos têm subconceito Y semelhante"). Não precisa visualização de mini-grafo ainda, mas texto estruturado. - **UI Local operando offline:** *Meta:* A interface web local carrega via servidor local, permitindo as interações acima. Done: Testado executando sem internet, todas funções (import, compute, explain) funcionam. - **Exportar PNG do heatmap:** *Meta:* Usuário pode clicar “Exportar PNG” e salvar a imagem do heatmap atual. Done: Abrir o arquivo exportado e ver que corresponde à visualização (mesma coloração e incluem rótulos legíveis). Pode ser restrito ao heatmap principal (não necessariamente sub-heatmap). - **Desempenho MVP:** Suportar grafos pequenos (~até 50 nós cada) com recálculo de heatmap < 2s e UI >30fps. Done: Teste manual com dataset de ~50x50.

Esses itens completam um MVP demonstrável. Critério de sucesso MVP: Um use-case simples (ex.: comparando analogias em um dataset acadêmico pequeno) é realizado end-to-end dentro do app local.

V1 (Versão 1.0 - Primeira versão completa): Após MVP, adicionar recursos-chave para abrangência e refinamento, mantendo robustez: - **Importação de Texto básica:** Permitir importar texto simples. Done: Usuário carrega um .txt com frases simples "X is Y of Z...", e o sistema produz um grafo com pelo menos alguns nós e relações correspondentes. (Mesmo que não seja perfeito, mas demonstra utilidade). - **Cache incremental:** Implementar caching para evitar recálculo total desnecessário. Done: Quando comparar os mesmos dois grafos sem alterações, resultado surge quase instantâneo da cache; se um grafo muda, apenas então recalca. Confirmado via logs ou timing (2a execução muito mais rápida). - **Feedback Humano integrado:** UI com botões Boa/Ruim funcionando e backend armazenando. Done: Sistema reage, por

exemplo, marcando visualmente ou gravando em DB. Opcionalmente, leve ajuste de peso: definiremos done se pelo menos feedback é persistido e pode ser visto (ex.: talvez uma seção "Meus feedbacks" ou via log). - **Melhoria no Explain (lista + mini-grafo)**: Expandir explicação com estrutura. Done: Painel de explicação agora lista claramente top-3 sub-matches e divergências. Critério: para um caso com subcomponentes, exibir esses pares filhos textual ou com simples gráfico (ex.: indented list or a tiny network image). Visual clarity improved. - **UI/UX Polimento**: Correções de usabilidade identificadas no MVP feedback. Done: Ex.: scroll de heatmap implementado, tooltips delay ajustado, etc. (Vago, mas teremos lista de issues a resolver do MVP). - **Documentação Básica**: Help ou documentação embutida para importadores (ex.: "como formatar seu JSON", ou "dicas para texto"). Done: um arquivo README ou seção help na UI cobrindo isso. - **Desempenho V1**: Suportar grafos médios (~200-500 nós) com otimizações. Done: Teste com ~200x200 grafos: UI continua responsiva (talvez usando top-K focusing by default). Podemos definir objetivo: heatmap calculado <10s para 200x200 e interação pan ~20fps com tiling básico.

V2 (Versão 2 - Recursos avançados e escala): Nesta fase, introduziremos capacidades mais avançadas e suporte a domínios extras: - **import_repo_{Python}**: Integrar plugin para código Python. Done: Usuário aponta para um repo médio (~seu 50 arquivos), a importação completa sem erros em tempo razoável (<5s), resultando em grafo de módulos/classes. Verificação: alguns nós correspondem a classes ou arquivos do repo, e arestas de dependência existem (ex.: se havia um import, virou aresta). - **Matching v2 (assignment otimizado)**: Substituir ou complementar algoritmo por correspondência ótima global. Done: Em datasets de teste onde guloso falhava (ex.: 2 nós competindo por 1), ver que V2 escolhe par ideal. E logs ou debug mostram Hungarian/ILP rodando. Critério: analogias globais mais consistentes, confirmadas por aumento de alguma métrica (ex.: uma pontuação F1 no benchmark). - **Modo "Comparar versões"**: Habilitar comparação de um grafo consigo mesmo em versões diferentes. Done: UI permite selecionar Graph A v1 e Graph A v2 e gerar um "diff de analogia". Critério: se diferenças existem, elas são destacadas – ex.: se um nó novo em v2 não tinha analogia em v1, agora aparece; ou color shift. Might present as heatmap of A_v1 vs A_v2 (which ideally should align on identical concepts - could highlight changed ones). At least demonstrate detection of added/removed nodes analogies. - **Ontologia/OWL (básico)**: Plugin import_ontology plenamente funcional. Done: Importar uma ontologia de ~100 classes não dá erro e produz grafo com hierarquia. E possivelmente, se compararmos com outro grafo, vemos analogias que fazem sentido (ex.: classes equivalentes map 1.0 if known). - **Desempenho e escala**: Otimizações WebGL/Tiling para suportar grandes grafos (~100k nós total). Done: Em um teste com ~1000x1000 analogias (1e6 cells), app não trava: utiliza LOD (talvez mostra agregação). Pan/zoom ainda possível a ~15fps com WebGL. Memory usage within limits (<2GB). Possibly gating: might not achieve full 100k by 100k explicit, mas at least handle 1000x1000 (1e6) gracefully via tiling. - **Refinamento UI**: e.g., grouping nodes by category or search highlight on heatmap. Done: New UI enhancements implemented as per feedback (like cluster axis or filter by tag). - **User testing & polish**: Before closing v2, get more user feedback (especially on new code importer, version compare) and refine accordingly. Done: All critical feedback from initial users resolved.

V3 (Versão 3 - Inovações e aprendizado): Esta versão explora funcionalidades de ponta e eleva a qualidade analítica: - **Embeddings estruturais / Learned matching**: Integrar um modelo ML local para melhorar analogias. Done: Usuário opta por baixar modelo (ou bundling a pre-trained small one), e sistema ao recalcular nota-se diferenças positivas. Critério: Em casos complexos (que requerem contexto global), V3 sugere analogias que V2 não via. Eg: emergir analogias cross-layer ou entre nodes sem tags óbvias. Quantitativo: slight improvement in benchmark metrics, qualitativo: domain expert approves new analogies. - **Superfícies 2D/3D (embedding + cor)**: Nova visualização onde cada nó de ambos grafos é projetado num plano comum ou espaço 3D, mostrando clusters e correspondências via proximidade e cor. Done: Modo opcional "Visualização em 2D": mostra um scatterplot onde nodes from both A and B appear

(maybe different shapes/colors) and analogous ones are near each other. If we color by domain or cluster by graph, user can see cross mixing. Or a 3D view using WebGL. Done criteria: For a known analogy set, the points of analogous pair appear near in embedding space. Could allow rotating 3D or so. It's exploratory, so done is a working interactive plot that adds insight (if not, can skip if time). - **Painéis de tendência temporal:** Se graphs têm versões com timestamp, oferecer um dashboard para ver como analogia global evolui. Ex.: a line chart of "similaridade global entre sistema e baseline X ao longo do tempo". Done: If we have data of multiple versions, the app can show e.g., "score médio" or count of analogias acima de threshold vs version/time. Or highlight which analogies appeared/disappeared in timeline. Done if an example with 3+ versions can be visualized as intended. - **Colaboração local (possível):** Perhaps allow multiple local users to share graphs (like through file sync) or multi-user feedback. Not explicitly in spec, skip if not needed. - **Revisão de Goodhart & 2^a ordem:** Implement advanced things like detect systematic optimizations (maybe outside scope, but V3 could incorporate a "anomaly" log). - **Performance & UX final polish:** Aim that even large uses (like corp ontology vs corp codebase) are feasible. Possibly integrate multi-thread for calc or partial GPU compute for embedding. UI finalize all small details (accessible nav, theming, etc.). - **Release documentation & community:** By V3 have thorough docs, maybe open source it or release widely.

Cada milestone deve satisfazer critérios objetivos: - Execução de todos testes relevantes (unitários, integrados) deve passar. - Validação com alguns usuários do target. - Documentação atualizada.

Resumindo: - MVP (Alpha) – funcionalidades básicas funcionando localmente. - V1 (Beta/1.0) – utilidade real com mais formatos e refinamento suficiente para uso contínuo. - V2 (1.5) – alcance ampliado (código, ontologias), matching mais inteligente, apto a problemas maiores. - V3 (2.0) – diferenciação por inteligência (embedding learning), visualizações inovadoras, produto maduro com feedback adaptativo sólido.

Transição entre fases ocorrerá conforme confiança: MVP internamente, V1 possivelmente primeira versão pública limitada, V2 broad adoption in advanced orgs, V3 consolidado para comunidade ampla.

L) Riscos e Mitigações

Listamos os principais riscos identificados e como mitigá-los:

- **Falsos Positivos "Quentes" (Analogias espúrias por semântica):** Risco de o score ser alto apenas porque dois conceitos têm nomes parecidos ou texto semelhante, mas na verdade sem analogia estrutural (ex.: "Java" (ilha) vs "Java" (linguagem)). Mitigação: A dimensão semântica é forte, mas exigimos confirmação estrutural mínima. Implementamos no cálculo que um par não será destacado se as demais dimensões forem muito baixas (trigger de alerta). Além disso, sinalizamos visualmente (Δ) casos de disparidade entre dimensões ³. O usuário é alertado e encorajado a examinar divergências no painel. Feedback do usuário também ajuda: se marcarem como "ruim", esse par será suprimido futuramente.
- **Mismatches de Escala (quinas por escala incompatível):** Exemplo: comparar um sistema todo com um subsistema específico – pode resultar analogia injustamente alta se não normalizar. Mitigação: Não comparar camadas muito distintas diretamente (ex.: só comparar layer 0 com layer 0, layer 1 com layer 1 na visualização principal). Se usuário força, avisar "Conceitos em níveis diferentes – analogia pode ser improdutiva". Também utilizar a dimensão temporal/escala: se escalas diferem muito, punir o score, que efetivamente impede alguns mismatches de aparecer no topo. Divergences listará "escala diferente" para conscientizar.

- **Goodhart's Law (Métrica otimizada de forma perversa):** Se equipes começarem a "jogar com o score" – por ex., ajustando o sistema observado apenas para maximizar analogia com um padrão, sem melhorar de fato o sistema. Mitigação: A ferramenta deve evitar ser uma métrica cega.
Inserimos *mecanismos de 2^a ordem*:
 - Logging de uso suspeito: se detectamos que a entrada do grafo está sendo alterada repetidamente de forma incremental apenas para aumentar score (ex.: adicionando tags irrelevantes), podemos emitir um aviso do tipo "⚠️ Cuidado: otimização dirigida ao score detectada – verifique se as mudanças fazem sentido real".
 - Educar no produto: Documentação e UI reforçam que a métrica é um auxílio, não objetivo final. Ex.: um tooltip no score global dizendo "Maximizar este número não garante melhor design - use analogias criticamente".
 - Modo revisão: Permitir um especialista desligar componentes do score para verificar robustez (ex.: recalcular sem semântica e ver se analogia ainda aparece). Isso pode revelar dependência exacerbada.
 - Em última instância, lembramos aos usuários (no onboarding ou docs) sobre a Lei de Goodhart: "quando uma medida vira meta, deixa de ser boa medida" ³, incentivando foco nas explicações e não apenas no número.
- **Explosão combinatória de subconceitos:** À medida que grafos crescem em tamanho e camadas, o número de comparações cresce quadraticamente. Um risco é o sistema ficar sobrecarregado ou a visualização virar "muro de pontos" ininterpretável. Mitigações:
 - Computacional: usar *lazy evaluation* e *filtragem*: por padrão, não calcular todas as combinações se grafo > N; em vez disso, calcular top-K ou sob demanda (ex.: calculamos grosso modo e refinamos quando usuário aproxima zoom de uma região). Já no MVP definimos limites e avisamos: "Matrix grande, exibindo top 100 correspondências - refine seleção".
 - Visual: Tiling e LOD como dito mantêm performance. Em termos de UX, fornecemos **ferramentas de foco** – filtragem por tag (ex.: "mostrar apenas analogias envolvendo nós marcados 'controle'"), ou highlight apenas top quartile. Também, drill-down ajuda a segmentar: o usuário olha por partes ao invés de tudo de uma vez.
 - Existe ainda a possibilidade de clusterizar nós e comparar grupos em vez de individuais (futuro).
 - Em logs, monitorar tempo do algoritmo: se ultrapassar um threshold, abortar graciosamente e sugerir usar filtros.
- **Bias de fontes (Ontologias ou Dados tendenciosos):** Se um grafo vem de uma fonte com um viés (ex.: uma ontologia feita por certo grupo que enfatiza certos conceitos), as analogias refletirão isso e podem induzir conclusões enviesadas. Mitigação:
 - Mostrar a proveniência: no painel Explain, podemos listar de qual origem vem cada peça de evidência (ex.: "Fonte: Ontologia X" para uma relação). Assim, o usuário pode ponderar a confiabilidade.
 - Permitir mix de fontes: incentivamos importar múltiplas fontes para contrabalancear (ex.: duas ontologias diferentes do mesmo domínio) e talvez fundi-las.
 - Internamente, podemos detectar se um grafo é altamente auto-referencial ou incompleto e então marcar sua confiança baixa (GraphSpec *confidence* meta). Se comparações envolvem um grafo de confiança baixa, sinalizar no UI ("⚠️ Grafo B é incompleto ou tendencioso, analogias podem faltar partes").
 - Lidar com bias de ML (quando embeddings vierem): usar modelos localmente controlados (e não black-box cloud) e manter o human-in-the-loop para revisar sugestões, reduz riscos de conclusões automáticas incorretas.

- **Falha em analogias críticas (falso negativo):** O risco oposto: a ferramenta deixar passar analogias importantes (score baixo erroneamente). Isso pode minar confiança. Mitigação:
- Permitir ao usuário procurar manualmente correspondências: ex., se suspeita que X ~ Y deveriam ser analógicos mas não apareceu, ele pode buscá-los (via search ou manual selection) e ver breakdown – talvez descobrir falta de dados e complementar.
- Melhorar cobertura de dimensões: V3 embeddings e feedback irão justamente reduzir falsos negativos aprendendo padrões indiretos.
- Log de analogias rejeitadas: opcional, mas para desenvolvimento, podemos logar se houve pares que tinham match em 1-2 dims mas foram filtrados. Pode ajudar a ajustar thresholds.
- **UI Overload / Complexidade:** Há risco de a interface ficar muito carregada (muitos controles avançados) e confundir usuários. Mitigação:
- Progressiva revelação: escondemos opções avançadas até o usuário acionar, mantendo UI principal limpa.
- Realizar testes de usabilidade e simplificar fluxos. Ex.: Talvez auto-drill quando analogia clara em vez de pedir clique.
- Documentação e exemplos integrados: talvez incluir um tutorial interativo leve para guiar primeiro uso.
- **Riscos técnicos:**
 - **Compatibilidade:** Browser differences (we target modern Chromium/Firefox likely, and maybe embed in Electron). Test cross env.
 - **Data corruption:** e.g., if app crashes while writing DB. Use robust operations (transactions). Mitigate by backups of DB or using append-only log for crucial data.
 - **Memory leaks:** continuous usage might accumulate. We'll do memory profiling and use tools like `weakref` in Python for caches if needed, and ensure event listeners cleaned on UI.
 - **Threading issues:** background jobs vs UI need sync. We'll design properly with async/await or job queue to avoid race conditions.
 - **Resistência a input malicioso:** Since it's local, less risk, mas se um arquivo importado tem estrutura estranha, ensure robust parsing (avoid code execution in importers, limit recursion). Possibly a risk if user imports code with malicious content – but we parse static, not run it. We'll double-check that (e.g., don't `import` the module, parse as text AST).

Para cada risco, implementamos *detectores* ou *sinais*: e.g., dimension variance for spurious analogies, usage pattern for Goodhart, etc., e correspondentes *mitigações ativas* (alertas, ajuste de peso, restrições) ou *passivas* (documentação, requiring user confirmation for certain actions). Também manteremos um log/debug mode para devs que registra quando certos mitigations dispararam, para ajustar thresholds ou false positives.

Exemplo de mensagem UX: "Analogias quentes porém arriscadas marcadas com ícone ⚠ (baseado em disparidade semântica vs estrutural). Revise explicações nesses casos." – Isso torna o usuário parte da mitigação, conferindo senso crítico.

No geral, ao tratar proativamente esses riscos, esperamos que a ferramenta permaneça útil e confiável mesmo em cenários adversos, e que os usuários sejam auxiliados a não cair em armadilhas cognitivas ou técnicas.

M) Checklist Final de Qualidade (Produto + Visualização)

Antes de liberar a versão final, verificamos todos os pontos a seguir, garantindo que o produto atenda aos padrões estabelecidos:

- [] **Local-first & Offline:** Aplicação funciona integralmente sem internet. Verificado que nenhuma funcionalidade realiza requisições externas (inspecionado via monitor de rede ou código). Dados do usuário (grafos, feedback) permanecem local e persistem em reinicializações.
- [] **Importadores funcionando:** Todos os plugins implementados (JSON, texto, ontologia, repo) foram testados com exemplos reais e documentados. Documentação de uso de cada importador disponível (no manual ou UI de ajuda). O sistema lida graciosamente com arquivos inválidos (mensagem de erro clara).
- [] **Modelo de dados consistente:** GraphSpec JSON salvo para cada grafo importado/ criado. IDs únicos e estáveis. Versionamento aplicável e testado (importar v2 de grafo não sobrescreve v1 sem aviso; é possível ter ambos). Exemplo de JSON de cada tipo de input incluído nos materiais de teste.
- [] **Heatmap principal correto:** Em casos de teste conhecidos, o heatmap exibe padrões esperados (verificado contra cálculos offline ou expectativas do especialista). Nenhum deslocamento ou eixo trocado – correspondências conhecidas aparecem na célula certa. Legenda de cor presente e interpretável.
- [] **Drill-down operante:** Clique duplo ou botão para detalhar subconceitos abre novo heatmap de subnível. Confirmado que eixos correspondem aos subconceitos dos nós selecionados e que voltar ao nível anterior funciona (estado anterior restaurado sem recomputar inutilmente).
- [] **Painel de Explicação informativo:** Para analogias testadas, painel lista pelo menos: tags comuns, uma relação suporte e uma divergência (quando aplicável). Formatação legível (títulos ou ícones para cada seção, ex: para relações, ⚡ para divergências). Nenhuma sobreposição de texto ou corte em resoluções padrão.
- [] **Interatividade 60fps (em alvo):** Teste manual ou instrumentado de pan e zoom do heatmap confirma animações suaves em grafos de tamanho médio. Nenhum lag perceptível ao arrastar. Tooltip aparece rapidamente e sem flicker. Hover highlighting não trava render.
- [] **Renderização HiDPI nítida:** Verificado em tela retina (ou simulada via DPR): textos de eixos e números visíveis sem borrão, linhas definidas. Export PNG em alta resolução corresponde (se DPR=2, exporta 2x tamanho para igual qualidade, ou de outra forma compensamos).
- [] **Colormap apropriado:** Checamos que default é perceptualmente uniforme (ex.: Viridis). Realizamos um teste convertendo a imagem em escala de cinza – variação de luminosidade continua representando os dados (indicando uniformidade). Confirmado ausência de espectro arco-íris distorcido. 2
- [] **Acessibilidade de cores:** Passamos o simulador daltônico (ferramenta) no heatmap – as diferenças ainda distinguíveis. Contraste de textos (ex.: label sobre fundo branco ou sobre tile claro) $\geq 4.5:1$.
- [] **Exportações funcionais:** Exportar PNG gera arquivo correto (abrível e igual ao visto na tela, fora talvez diferenças de fonte se não embedar – idealmente rasterizado). Se implementado, export SVG não apresenta elementos faltando. Export JSON/relatório contém dados consistentes (ex.: lista de top analogias com scores e contextos).
- [] **Cache & incremental:** Repetir comparações não recalcula (verificado via logs ou timestamp). Modificar um grafo ligeiramente e recomparar resulta em atualização parcial ou, no mínimo, marcação de stale e recálculo só uma vez. Nenhum bug de usar cache antigo erroneamente.

- [] **Persistência & Reabrir:** Após fechar e reabrir a aplicação, grafos importados anteriormente ainda disponíveis. Última comparação reaparece se implementado (ou pelo menos, grafos lembrados e fácil re-trigger). Feedback do usuário dado em sessão anterior ainda registrado (visível talvez em algum log ou efeito).
- [] **Logs e Debug:** Sistema gera logs (no console ou arquivos) sem informações sensíveis. Log padrão informa apenas ações e tempos. Em modo debug, logs detalham mapeamentos, mas mesmo assim usuário consciente (tem que ativar). Confirmamos que habilitar debug não quebra performance ou funcionalidade.
- [] **Resiliência a erros:** Tentamos operações inválidas (importar arquivo corrompido, clicar compare sem grafos selecionados, etc.) e app não crasha. Mensagens de erro exibidas orientam o usuário ("Arquivo não pôde ser lido, verifique formato"). Nenhum stacktrace cru exibido ao usuário.
- [] **Testes automatizados passando:** Todos unitários e integrados implementados rodam com sucesso no CI. Cobertura de código satisfatória em módulos críticos (score calc, import).
- [] **Benchmark performance respeitado:** Em datasets de benchmark (mini e médio), confirmamos métricas dentro do planejado (tempos, FPS, etc. conforme seção J). Especialmente, teste com ~1000 nós por grafo não leva horas ou travamentos (se for pesado, degrade de forma controlada ou avise).
- [] **Mitigações de risco ativas:** Induzimos cenários de cada risco e confirmamos a mitigação:
- Ex.: para falso hot, criamos dois nós com mesmo label sem relação – o sistema marcou Δ ou deu baixa pontuação devido às outras dimensões nulas.
- Para Goodhart: alimentamos feedback extremo e verificamos que o sistema não fez algo absurdo (como atribuir peso 0 a semântica sem aviso).
- Out-of-memory: testamos import muito grande, app recusou ou pediu confirmação em vez de travar.
- [] **UX final polido:** Verificação subjetiva: interface está coerente, sem elementos desalinhados. Ícones e textos traduzidos/localizados (Português no nosso caso) corretamente. Tooltip content and any help text revisado linguisticamente.
- [] **Documentação & Roadmap entrega:** PRD finalizado (este doc), guia de usuário e possivelmente um changelog/roadmap visível para usuários avançados entenderem futuras expansões (transparência).
- [] **Prontidão de release:** Instalador ou pacote preparado (se for desktop) ou instruções simples para rodar local. Testado em pelo menos 2 plataformas (Windows e macOS/Linux).

Cada item dessa lista deve ser ticado antes da versão release (v1.0 ou v2.0 conforme scope). Isso garante que não negligenciamos nada crucial, entregando um produto robusto, performático e realmente útil no que se propõe.

Após cumprir tudo acima, estaremos confiantes de “ganhar o jogo” – a ferramenta permitirá ao usuário explorar analogias complexas rapidamente, com uma experiência de qualidade de produto top-tier, e fundamentos sólidos para evolução futura.

1 Local-First Software

<https://lofi.so/>

2 Color Map Advice for Scientific Visualization | Kenneth Moreland

<https://www.kennethmoreland.com/color-advice/>

3 Entre Indicadores e a Realidade: métricas não mudam resultados, só comportamentos | by Manoel Medeiros | Nov, 2025 | Medium

<https://manoelmedeiros.medium.com/entre-indicadores-e-a-realidade-m%C3%A9tricas-n%C3%A3o-mudam-resultados-s%C3%B3-comportamentos-e9e9a9d10983>

4 Ferramentas para Visualizaçao de Informaes

<https://www.sigapp.org/sac/sac2007/T3.pdf>

5 Unlocking Insights with Visual- Heatmap: A High-Performance JavaScript Module for Heatmap

Visualisation | by Narayana swamy | Medium

<https://medium.com/@narayanaswamy14/unlocking-insights-with-visual-heatmap-a-high-performance-javascript-module-for-heatmap-ce83c21abbb9>