Capítulo 3

Agentes Inteligentes para Configuração de Redes de Computadores: Da Teoria à Prática com LLM, SLM, RAG e IA Agentic

William L. Reiznautt (UNICAMP), Eduardo Cerqueira (UFPA),
Diogo M. da Cunha (UNICAMP), Leandro A. Villas (UNICAMP),
Antonio A. F. Loureiro (UFMG), Denis Rosário (UFPA),
Allan M. de Souza (UNICAMP), Nelson L. S. da Fonseca (UNICAMP)

Abstract

Network management has become increasingly complex due to the convergence of cloud, edge, 5G/O-RAN, and multivendor environments. This chapter presents intelligent agents for network configuration in the context of NetOps 2.0, integrating LLMs, SLMs, RAG, and Agentic AI. It discusses multimodel architectures, the SLM-first strategy, and reasoning paradigms such as ReAct, Pre-Act, and Structured Cognitive Loop. The MCP and A2A protocols are also examined, along with their security challenges. In the practical section, the chapter describes the implementation of an agent using Python, Ollama, and OpenWebUI for Linux network automation, including bridges, VLANs, IPv4/IPv6 routing, and VXLAN under the cycle Interpret→Plan→Execute→Validate→Correct. A case study and final benchmarks highlight paths toward autonomous networks.

Resumo

O gerenciamento de redes tornou-se complexo diante da convergência entre nuvem, borda, 5G/O-RAN e ambientes multivendedor. Este capítulo apresenta agentes inteligentes para configuração de redes no contexto do NetOps 2.0, integrando LLMs, SLMs, RAG e IA Agentic. São discutidas arquiteturas multimodelo, a estratégia SLM-first e paradigmas como ReAct, Pre-Act e Structured Cognitive Loop. Também são analisados os protocolos MCP e A2A e seus desafios de segurança. Na parte prática, descreve-se a implementação de um agente com Python, Ollama e OpenWebUI para automação de redes Linux, incluindo bridges, VLANs, roteamento IPv4/IPv6 e VXLAN no ciclo Interpretar→Planejar→Executar→Validar→Corrigir. Um estudo de caso e benchmarks finais apontam caminhos para redes autônomas.

3.1. Introdução

As redes de telecomunicações evoluíram para sistemas de larga escala altamente complexos, abrangendo desde o núcleo da rede (*core*) e o transporte até a borda (*edge*) e as redes de acesso via rádio (RAN) [Zhou et al. 2024]. Essa infraestrutura moderna é caracterizada por ambientes híbridos que integram recursos físicos (*bare metal*), virtualização de funções de rede (*Network Function Virtualization* – NFV), contêineres e Redes Definidas por Software (*Software Defined Network* – SDN) [Huang et al. 2023].

A natureza heterogênea e multivendedor dessas redes exige que os operadores gerenciem uma ampla variedade de dispositivos de diferentes fornecedores, cada um com suas próprias linguagens de configuração, protocolos e manuais técnicos [Huang et al. 2023, Zhou et al. 2024]. Além disso, a chegada do padrão de telefonia celular 5G e as projeções para o 6G introduzem desafios sem precedentes de escalabilidade, com a necessidade de suportar conectividade massiva, estimada em até 10 milhões de dispositivos por km^2, bem como prover comunicação com latência de submilissegundo [Boateng et al. 2024b]. A integração de múltiplos domínios, como os segmentos satelital, aéreo e terrestre, amplia ainda mais a complexidade operacional e a carga de gerenciamento dessas redes [Zhou et al. 2024].

As abordagens convencionais de Gerenciamento de Redes e Serviços (*Network and System Management* – NSM), baseadas em *scripts*, regras fixas e algoritmos estáticos, têm se mostrado inadequadas para lidar com a volatilidade e a complexidade das redes modernas. Essas soluções carecem de flexibilidade para se adaptar a mudanças na infraestrutura em tempo real e apresentam dificuldades no tratamento de dados não estruturados, como *logs* complexos e intenções expressas em linguagem natural. O processo de configuração manual é frequentemente descrito como trabalhoso, propenso a erros e dispendioso, sendo que uma única falha na configuração de uma lista de controle de acesso (*Access Control List* – ACL) pode causar interrupções graves na rede [Huang et al. 2023, Zhou et al. 2024]. Diante desse cenário, emerge o conceito de NetOps 2.0, também referido como inteligência de rede unificada, que foca na automação, agilidade e análise de dados para substituir processos manuais e estáticos [Huang et al. 2023]. O paradigma de NetOps 2.0 alinha-se ao conceito de *Zero-touch Network & Service Management* (ZSM), no qual a rede busca alcançar capacidades de auto-operação, auto-manutenção e auto-otimização, com intervenção humana mínima ou nula [Lira et al. 2024]. Nesse contexto, consolida-se a chamada "Era da Comunicação por Agentes", em que entidades inteligentes utilizam raciocínio e percepção para orquestrar ferramentas e protocolos de forma autônoma [Kong et al. 2025, Derouiche et al. 2025].

A evolução de Modelos de Linguagem de Grande Escala (do inglês, *Large Language Models* - LLMs), Modelos de Linguagem Pequenos (do inglês, *Small Language Models* - SLMs) e técnicas como Geração Aumentada (do inglês, *Retrieval-Augmented Generation* - RAG) por Recuperação amplia as capacidades de NetOps 2.0, permitindo raciocínio sobre estados de rede, integração com ferramentas externas e execução de ações de forma controlada [Derouiche et al. 2025, Chowa et al. 2026]. Estudos recentes [Long et al. 2025, Boateng et al. 2024b] mostram que esses modelos de IA aplicados a NetOps 2.0 são capazes de raciocinar sobre configurações, documentação técnica e estados complexos de sistemas, além de interagir com ferramentas externas e executar

ações de forma autônoma ou semi-autônoma. Nesse contexto, agentes de IA tornam-se particularmente relevantes para a área de redes de computadores, pois são capazes de interpretar solicitações de alto nível expressas em linguagem natural, consultar bases de conhecimento estruturadas ou não estruturadas, planejar sequências de ações e executar comandos de maneira validada e controlada.

Neste minicurso, serão apresentados os conceitos para projetar e implementar agentes inteligentes para automação e configuração de redes de computadores, com foco em arquiteturas práticas e reprodutíveis. A abordagem adotada no minicurso privilegia o uso de SLMs e LLMs executados localmente, a integração com ferramentas externas e a definição de fluxos de execução seguros, permitindo a experimentação em ambientes controlados antes da generalização para cenários reais mais complexos. Como estudo de caso prático, o documento explora a construção de agentes para automação de redes em ambientes Linux, utilizando *iproute2* e *network namespaces* como uma plataforma controlada e reprodutível para experimentação e compreensão do funcionamento desses agentes. Ao final do minicurso, os participantes estarão capacitados a integrar conceitos de IA e administração de sistemas, projetando agentes aplicáveis a diferentes ambientes e cenários de automação inteligente de redes de computadores.

O restante deste documento está organizado da seguinte forma. A Seção 3.2 apresenta os fundamentos de agentes inteligentes e modelos de linguagem. A Seção 3.3 discute arquiteturas de agentes com SLMs e LLMs. A Seção 3.4 introduz o paradigma de IA Agêntica e seu ciclo deliberativo. A Seção 3.5 aborda a engenharia de *prompts*. A Seção 3.6 explora o uso de RAG para aumento de precisão. A Seção 3.7 descreve a arquitetura prática baseada em Python, Ollama e OpenWebUI. A Seção 3.8 detalha a implementação do agente configurador. A Seção 3.9 discute aspectos de segurança. Por fim, a Seção 3.10 apresenta as conclusões.

3.2. Agentes Inteligentes e Modelos de Linguagem

Diferentemente dos sistemas tradicionais de NSM, baseados em regras estáticas e fluxos pré-definidos, as abordagens convencionais apresentam limitações diante da complexidade, dinamicidade e escala das redes modernas. Nesse contexto, consolida-se o paradigma de NetOps 2.0, no qual a operação da rede passa a incorporar mecanismos capazes de interpretar objetivos de alto nível, tomar decisões com base no contexto e adaptar seu comportamento ao longo do tempo. Essa mudança desloca o foco da configuração manual para a definição de intenções, viabilizando uma operação mais declarativa e orientada a resultados. Destacam-se três capacidades fundamentais: interpretação de intenção (*Intent-Driven*), planejamento automático e diagnóstico inteligente.

A Interpretação de Intenção é responsável por traduzir objetivos de alto nível, expressos por operadores ou por requisitos de negócio, em metas técnicas concretas para a rede [Boateng et al. 2024b]. Essa capacidade reduz a complexidade administrativa ao aproximar a linguagem humana das configurações operacionais, permitindo que a infraestrutura atue de acordo com políticas orientadas a resultados [Lira et al. 2024]. Em vez de o operador precisar especificar manualmente cada comando e cada dependência técnica, torna-se possível expressar a finalidade desejada, cabendo ao sistema interpretar como esse objetivo deve ser materializado.

No planejamento automático, uma vez definida a intenção, a rede deve determinar quais ações precisam ser executadas para satisfazê-la. Esse processo é essencial em cenários que exigem raciocínio multietapa, coordenação de recursos e adaptação contínua, como no fatiamento de rede (*network slicing*), na alocação dinâmica de recursos e na orquestração de serviços sob demanda [Boateng et al. 2024b, Zhou et al. 2024]. Em termos práticos, essa capacidade permite decompor uma solicitação ampla em uma sequência coerente de subtarefas, respeitando dependências técnicas, restrições de segurança e o estado corrente da infraestrutura.

Por fim, no diagnóstico inteligente, após a execução, a rede deve ser capaz de identificar desvios, falhas e degradações de desempenho. Em ambientes distribuídos e de larga escala, cada vez mais comum nos dias atuais, o *troubleshooting* manual tende a ser lento, custoso e sujeito a erro. Nesse contexto, técnicas baseadas em LLMs e agentes inteligentes podem apoiar a análise de *logs*, a correlação de eventos, a interpretação de telemetria e a inferência de causas prováveis, acelerando a detecção e a resolução de problemas [Huang et al. 2023, Boateng et al. 2024b].

Em conjunto, essas capacidades formam um ciclo de operação inteligente para o NetOps 2.0, no qual a rede compreende a intenção, planeja as ações necessárias, executa adaptações e diagnostica eventuais falhas para corrigir seu comportamento. A integração entre IA generativa e agentes autônomos desponta como um caminho promissor para transformar as infraestruturas de rede em sistemas mais resilientes, responsivos e adaptativos [Boateng et al. 2024b].

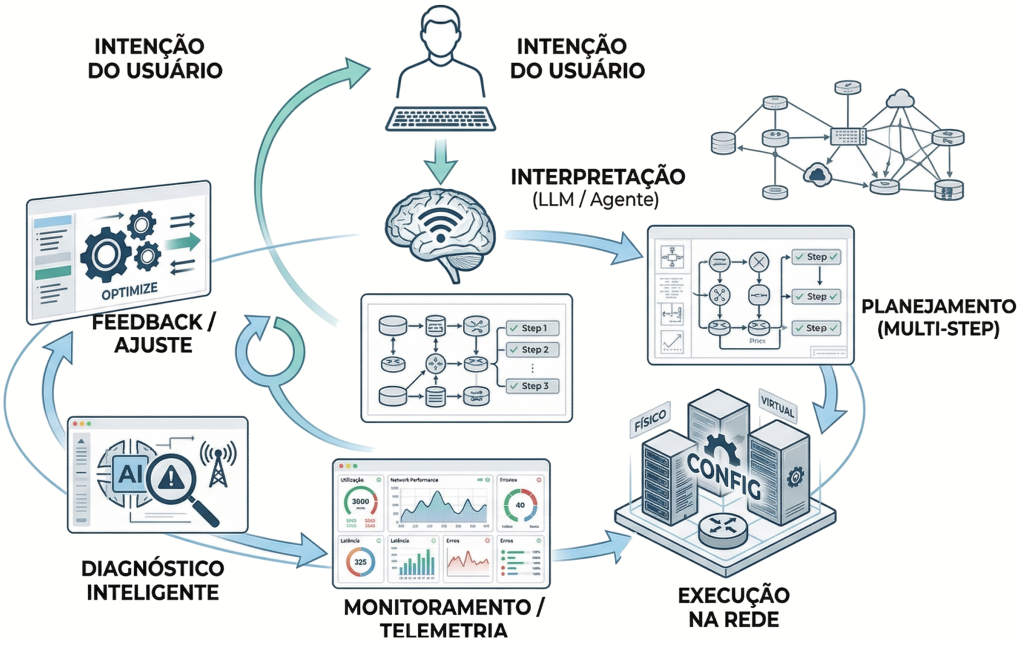

Figura 3.1. Ciclo de processamento de IA em redes

A Figura 3.1 apresenta um exemplo do ciclo de processamento de IA em redes envolvendo desde a intenção do usuário até a execução de operações na rede. Mais do que um conjunto abstrato de ideias, esse ciclo estabelece a base operacional dos sistemas agênticos aplicados à rede. Na prática, ele pode ser implementado como um laço iterativo

entre interpretação, planejamento, execução e validação, no qual o modelo de linguagem atua como núcleo cognitivo, enquanto ferramentas externas executam ações e devolvem observações sobre o estado do ambiente. Essa visão será retomada nas próximas seções, culminando na implementação prática de um agente configurador de redes.

3.2.1. Ecossistema de Modelos de IA Generativa

Em vez de depender de um único modelo monolítico, a arquitetura moderna de sistemas inteligentes distribui responsabilidades entre componentes especializados, buscando modularidade, determinismo, eficiência e manutenibilidade [Bandara et al. 2025]. Essa abordagem permite que diferentes modelos assumam papéis distintos em cada subetapa de uma tarefa complexa, formando um ecossistema cooperativo no qual cada interação faz parte de um fluxo maior de decisão e execução. No contexto de redes de computadores, essa organização é particularmente relevante. Tarefas como interpretar uma solicitação, consultar documentação, planejar mudanças, validar comandos, executar ações e verificar resultados possuem requisitos distintos de latência, precisão, custo e auditabilidade. Por essa razão, sistemas agênticos modernos podem empregar diferentes tipos de modelos especializados, entre os quais se destacam:

- **Modelos Base (*Fast Models*):** são modelos menores ou destilados, otimizados para baixa latência e custo reduzido. Em ambientes de rede, são adequados para tarefas rotineiras, como classificação de intenção, extração de parâmetros de configuração e identificação rápida do tipo de solicitação recebida [OpenAI 2025].
- **Modelos de Raciocínio (*Reasoning Models*):** são voltados a capacidades cognitivas superiores e utilizam estratégias como *Chain-of-Thought* (CoT) ou *Tree-of-Thought* (ToT) para decompor problemas em etapas lógicas. Em redes, são úteis para *troubleshooting* multietapa, planejamento de alterações que envolvem vários dispositivos e análise de dependências entre ações [Yao et al. 2023].
- **Modelos Agentes (*Tool-Using Models*):** atuam como núcleo cognitivo do fluxo de trabalho, sendo treinados ou configurados para decidir quais APIs, comandos ou ferramentas externas devem ser invocados, em que momento isso deve ocorrer e como os resultados devem ser incorporados ao processo decisório [Schick et al. 2023]. No domínio de NSM, isso inclui execução de comandos via SSH, consulta a sistemas de monitoramento, controladores SDN, inventários ou mecanismos de validação.
- **Modelos de *Embeddings*:** são fundamentais para converter documentos textuais, registros operacionais e até artefatos visuais em representações vetoriais, viabilizando busca semântica e recuperação eficiente de conhecimento [Park et al. 2023]. Em redes, permitem localizar trechos relevantes de manuais, *playbooks*, RFCs, incidentes passados e configurações históricas.
- ***Verifier / Critic*:** atuam como auditores de qualidade, verificando as saídas produzidas por outros modelos para identificar alucinações, erros de sintaxe, violações de política ou inconsistências operacionais [Zhou et al. 2024]. Em cenários de rede, podem ser associados a validadores sintáticos, simuladores, políticas internas ou ferramentas como Batfish.

- ***Router / Orchestrator*:** funcionam como coordenadores do fluxo de trabalho, selecionando o modelo ou recurso mais adequado para cada subtarefa [Lira et al. 2024]. Em uma arquitetura de rede, esse componente pode decidir quando usar um SLM local de baixa latência e quando escalar o problema para um LLM mais robusto.
- ***Memory Models*:** gerenciam retenção de informações em diferentes escalas temporais, combinando memória de curto prazo, associada à janela de contexto, com memória de longo prazo, normalmente baseada em armazenamento externo, como bancos vetoriais ou históricos operacionais [Park et al. 2023]. Em NSM, isso permite incorporar topologias, estados anteriores, incidentes passados e decisões já tomadas.

O sistema apresentado neste minicurso materializa essa organização em um cenário de redes, no qual cada componente assume um papel específico no ciclo operacional. Modelos rápidos podem ser usados para classificação de intenção e extração de parâmetros. Modelos de raciocínio podem apoiar a decomposição de tarefas complexas. Modelos agentes podem interagir com ferramentas externas. Finalmente, módulos verificadores podem garantir a conformidade das ações antes da aplicação. Essa arquitetura modular aproxima os conceitos de IA generativa das práticas reais de NSM, além de facilitar a substituição ou evolução de componentes conforme os requisitos de desempenho e segurança [Lira et al. 2024].

As distinções entre as diferentes tecnologias de IA generativa são frequentemente confundidas, embora desempenhem papéis bastante distintos em um ecossistema inteligente. Essa distinção é particularmente importante em redes, nas quais não basta gerar texto tecnicamente plausível. É necessário compreender objetivos, consultar contexto, decidir ações, interagir com ferramentas e validar resultados antes de qualquer aplicação sobre a infraestrutura.

Os **chatbots** são aplicações voltadas à interação conversacional, nas quais o usuário fornece um *prompt* e o sistema retorna uma resposta em um ciclo simples de pergunta e resposta [Bandara et al. 2025]. Sua capacidade de planejamento tende a ser inexistente ou depende inteiramente da condução manual do usuário, já que não decompõem tarefas por conta própria. Sua execução também é limitada, restringindo-se essencialmente à geração de texto. Quando utilizam ferramentas externas, isso ocorre de forma passiva ou bastante restrita, sem orquestração autônoma. No domínio de redes, seu uso é mais apropriado para suporte informacional, como explicar comandos, descrever protocolos ou interpretar mensagens de erro, mas não para operar diretamente a infraestrutura [OpenAI 2025, Kong et al. 2025].

Os **LLMs** são modelos treinados em grandes volumes de dados textuais para compreender e gerar linguagem humana, funcionando como núcleo cognitivo de muitos sistemas de IA [Kong et al. 2025, de Lamo Castrillo et al. 2025]. Embora sua operação fundamental esteja baseada na previsão do próximo *token*, esses modelos apresentam capacidades emergentes de raciocínio e lógica, o que lhes confere potencial para planejamento, por exemplo, por meio de estratégias como *Chain-of-Thought* [Kong et al. 2025, Du et al. 2026]. Em redes, podem gerar configurações, sugerir comandos, resumir logs ou apoiar o diagnóstico de falhas. No entanto, sua atuação isolada continua predominantemente reativa. Mesmo quando integrados a mecanismos de *function*

calling, os LLMs puros normalmente não mantêm controle autônomo contínuo do fluxo operacional, dependendo de um sistema externo para decidir quando executar, validar ou corrigir ações [Schick et al. 2023, Kong et al. 2025].

Por outro lado, os **SLMs** são versões compactas de modelos de linguagem, frequentemente obtidas por técnicas como destilação de conhecimento a partir de modelos maiores [Sanh et al. 2020]. Sua principal vantagem está na eficiência computacional, com menor latência e menor custo, o que favorece sua execução em dispositivos de borda ou ambientes com recursos limitados [Sanh et al. 2020, Zhou et al. 2024]. Quando ajustados para domínios específicos, como NSM, podem apresentar desempenho elevado em tarefas estruturadas, como geração de comandos de configuração, validação de sintaxe, classificação operacional e execução de fluxos padronizados de automação. Em aplicações com uso de ferramentas, SLMs podem alcançar desempenho comparável ou até superior ao de LLMs em *tool calling* e *function calling*, com custo significativamente menor, da ordem de 10 a 100 vezes [NVIDIA Research 2025]. Sua autonomia tende a ser de média a alta em tarefas repetitivas e especializadas, embora permaneça mais restrita fora de domínios estreitos.

Por fim, os **agentes inteligentes** representam uma evolução em que o modelo de IA deixa de atuar apenas como gerador de respostas e passa a desempenhar o papel de uma entidade capaz de agir sobre o ambiente [Kong et al. 2025]. Nesses sistemas, o modelo é utilizado para interpretar objetivos, decompor tarefas, construir planos, invocar ferramentas, monitorar resultados e ajustar suas decisões dinamicamente conforme o contexto. Sua execução é ativa, pois o agente não apenas produz texto, mas realiza ações concretas no ambiente digital ou físico por meio de ferramentas e serviços externos. Em redes, isso significa consultar telemetria, recuperar documentação, gerar comandos, executar ações por SSH, API ou NETCONF, verificar resultados e decidir se a tarefa foi concluída ou se requer correção. Essas características se apoiam em quatro propriedades fundamentais: percepção, memórias de curto e longo prazo, raciocínio ou planejamento e capacidade de ação [Kong et al. 2025, Elkael et al. 2026].

A Tabela 3.1 sintetiza essas diferenças em termos de planejamento, execução, uso de ferramentas e autonomia. Essa distinção é crucial para o entendimento do NetOps 2.0, no qual o objetivo não é apenas disponibilizar uma interface conversacional, mas empregar sistemas capazes de planejar, decidir e executar configurações de rede de forma controlada e progressivamente mais autônoma [Elkael et al. 2026, Lira et al. 2024].

3.2.2. Classificação por Tamanho

Os LLMs são caracterizados por possuírem bilhões ou até trilhões de parâmetros, sendo treinados em escalas massivas de dados para atingir capacidades versáteis de compreensão e raciocínio [Huang et al. 2023]. Modelos icônicos, como o GPT-3, possuem cerca de 175 bilhões de parâmetros, enquanto estimativas não confirmadas atribuem ao GPT-4 uma escala ainda superior [Bommasani et al. 2022]. Por outro lado, os SLMs são versões compactas e otimizadas, muitas vezes derivadas por destilação de modelos maiores, resultando em modelos mais leves e rápidos em tempo de inferência [Sanh et al. 2020, Bommasani et al. 2022]. A geração de modelos a partir de 2025 avançou significativamente além dos primeiros SLMs, com modelos como Phi-4-mini,

Tabela 3.1. Comparação simplificada entre chatbots, LLMs, SLMs e agentes

Categoria	Planejamento	Execução	Ferramentas	Autonomia
Chatbot	Manual / inexistente	Texto	Limitado	Baixa
LLM	Potencial (CoT)	Texto / código	Reativo (*function calling*)	Baixa
SLM	Alta em domínio específico	Rápida / estruturada	Ativo, com baixo custo	Média
Agente	Alta e dinâmica	Ativa, com ações sobre o ambiente	Orquestrado	Alta

LFM2.5, Qwen-2.5-7B e Llama-3.2 demonstrando que o limiar de capacidade necessário para tarefas agênticas estruturadas está abaixo do que se estimava anteriormente [Microsoft Research 2025, Liquid AI 2025, NVIDIA Research 2025]. A Tabela 3.2 apresenta um sumário de modelos SLM relevantes para agentes a partir de 2025.

Tabela 3.2. Modelos SLM relevantes para agentes a partir de 2025

Modelo	Parâmetros	Destaque para Agentes
Microsoft Phi-4-mini	3,8B	Raciocínio, *coding* e *function-calling* [Microsoft Research 2025]
Liquid AI LFM2.5	1,2B	Arquitetura não-*Transformer*; agentes *edge* e raciocínio *on-device* [Liquid AI 2025]
Qwen-2.5-7B	7B	Desempenho balanceado e bom suporte a *tool calling*
Llama-3.2-1B/3B	1–3B	Implantação ultra-leve em *mobile* e IoT
Ministral-3B/8B	3–8B	Especialização em chamadas de ferramentas

A distinção entre essas classes de modelos pode ser analisada a partir de quatro pilares principais. O primeiro é o **número de parâmetros**. Enquanto LLMs de ponta operam na escala de centenas de bilhões ou mais, os SLMs tendem a situar-se abaixo da faixa de 10 bilhões de parâmetros, priorizando eficiência sem abandonar completamente a capacidade de generalização [Boateng et al. 2024b]. O segundo pilar é o **contexto**. LLMs modernos oferecem janelas de contexto muito amplas, permitindo processar grande volume de documentação, histórico operacional e instruções em uma única interação. SLMs tendem a trabalhar com contextos menores, embora versões recentes tenham ampliado essa capacidade [Chowa et al. 2026]. Em redes, isso influencia diretamente o quanto de topologia, política, documentação e estado da sessão pode ser analisado de uma só vez. O terceiro pilar é o **custo**. O treinamento de um LLM de fronteira é proibitivo

para a maior parte das organizações, e sua inferência também tende a ser significativamente mais cara [Zhou et al. 2024]. Em contrapartida, SLMs apresentam custos mais acessíveis e tornam-se especialmente atrativos para tarefas rotineiras, frequentes e bem delimitadas [Zhou et al. 2024, Chowa et al. 2026]. O quarto pilar é a **infraestrutura**. LLMs normalmente exigem grande disponibilidade de GPUs e alto consumo energético para treinamento e inferência, sendo frequentemente inviáveis para execução contínua em ambientes locais ou distribuídos [Zhou et al. 2024]. Já os SLMs são mais adequados para executar em hardware comum, servidores de borda ou até dispositivos com recursos limitados [Sanh et al. 2020].

A escolha entre processar tarefas de IA na nuvem ou localmente envolve um compromisso entre desempenho, latência, privacidade e custo. Modelos em nuvem oferecem acesso direto a LLMs de maior capacidade, com abundância de recursos computacionais para tarefas complexas [Zhou et al. 2024]. No entanto, também apresentam maior latência de ponta a ponta, custos associados ao uso contínuo da API e riscos decorrentes do envio de dados sensíveis da infraestrutura para ambientes externos [Chowa et al. 2026]. Em contraste, modelos locais, seja na borda ou *on-device*, são mais adequados para aplicações que exigem resposta rápida, maior controle sobre os dados e menor dependência de conectividade [Zhou et al. 2024]. Em ambientes de rede, essa diferença é crítica, pois o tempo de resposta e a confidencialidade das informações operacionais podem ser fatores determinantes.

Dessa forma, a tendência mais promissora aponta para arquiteturas híbridas, nas quais a nuvem é utilizada para tarefas de maior complexidade e planejamento de alto nível, enquanto modelos locais, tipicamente menores, são responsáveis pela inferência rotineira e por tarefas específicas de domínio [Chowa et al. 2026, Zhou et al. 2024]. Esse arranjo é particularmente aderente ao gerenciamento de redes, onde a maior parte das operações é repetitiva e previsível, mas uma parcela menor demanda raciocínio mais profundo e maior poder de generalização.

Tese SLM-First e o Novo Paradigma Operacional

A execução local de modelos de larga escala impõe desafios significativos em termos de custo computacional, consumo de energia e requisitos de infraestrutura. Por outro lado, o uso contínuo de modelos em nuvem também envolve custos operacionais relevantes, como cobrança por requisição, latência de comunicação e transferência de dados. Nesse contexto, pesquisas recentes defendem que SLMs na faixa de 1 a 12 bilhões de parâmetros são suficientemente poderosos, mais adequados e economicamente mais viáveis para a maioria das invocações em sistemas agênticos [NVIDIA Research 2025]. Isso ocorre porque grande parte das tarefas executadas por agentes é simples, repetitiva e estruturalmente bem definida, como classificação de intenção, extração estruturada, roteamento, preenchimento de argumentos e chamadas de função. Tais tarefas não exigem, necessariamente, a capacidade de raciocínio aberto de modelos *frontier*. Nesses casos, SLMs ajustados para domínios específicos tendem a gerar saídas mais aderentes a formatos estruturados do que LLMs maiores, reduzindo a necessidade de pós-processamento e o número de *retries*, a um custo 10 a 100 vezes menor [NVIDIA Research 2025].

Nesse cenário, consolidam-se os padrões **SLM-default** e **LLM-fallback** [Microsoft Research 2025]. Nesse modelo, as requisições são inicialmente roteadas para um SLM rápido, frequentemente executado localmente. Em seguida, um módulo verificador avalia a qualidade, a confiança ou a conformidade da saída produzida. Caso o resultado seja insuficiente, a tarefa é escalonada para um LLM mais robusto, normalmente disponível em infraestrutura mais poderosa ou em nuvem. Essa abordagem reduz significativamente custos e latência, sem abrir mão da capacidade de lidar com casos complexos ou fora da distribuição do domínio rotineiro.

As métricas de engenharia associadas ao uso de SLMs em produção refletem objetivos práticos, como custo por tarefa bem-sucedida, taxa de validade de *schema*, taxa de chamadas executáveis, latências p50 e p95[1] e energia por requisição [NVIDIA Research 2025]. Em ambientes de rede, essas métricas são particularmente úteis porque aproximam a avaliação do modelo de critérios operacionais reais, e não apenas de *benchmarks* genéricos de linguagem.

3.2.3. Limitações

Apesar do potencial transformador, a adoção de modelos de linguagem e agentes inteligentes em redes enfrenta barreiras técnicas e operacionais que devem ser tratadas com rigor. A seguir são apresentadas algumas limitações, a saber: alucinações, risco operacional e dependência de contexto.

Alucinações correspondem à geração de informações factualmente incorretas, sem sentido ou fictícias, frequentemente apresentadas com elevado grau de confiança [Zhou et al. 2024, Lira et al. 2024]. Os modelos podem inventar fatos, embelezar conhecimento incompleto ou produzir associações técnicas incorretas com aparência plausível [Park et al. 2023]. Em infraestruturas críticas, como telecomunicações e redes corporativas, esse comportamento compromete diretamente a confiabilidade das soluções produzidas.

Não-determinismo deve-se a natureza probabilística dos LLMs implica que estados idênticos da rede podem produzir ações ligeiramente diferentes em execuções sucessivas [Elkael et al. 2026]. Essa variabilidade afeta a consistência e a previsibilidade dos planos de execução, dificultando seu uso em funções críticas ou fortemente auditadas [de Lamo Castrillo et al. 2025]. Embora parâmetros como temperatura permitam controlar parcialmente a variabilidade, o modelo continua operando sob uma distribuição probabilística, e não sob uma lógica estritamente determinística [Ha and Schmidhuber 2018].

Risco Operacional deve-se ao fato que a integração de LLMs pode introduzir pontos únicos de falha, já que erros, vieses ou limitações do modelo base tendem a se propagar para aplicações derivadas [Bommasani et al. 2022]. Além disso, a opacidade desses sistemas limita sua aceitação em contextos que exigem explicabilidade e previsibilidade [Lira et al. 2024, Boateng et al. 2024b]. O uso de modelos em nuvem também impõe riscos à privacidade e à segurança, uma vez que demanda o compartilhamento de

[1]Latências p50 e p95 são métricas de percentil usadas para medir o tempo de resposta de sistemas (e.g., APIs e bancos de dados). O p50 (mediana) representa o tempo em que 50% das solicitações são concluídas, indicando o cenário típico. O p95 (percentil 95) indica o tempo para 95% das solicitações, mostrando o desempenho dos casos extremos.

dados potencialmente sensíveis da infraestrutura [Lira et al. 2024]. Em cenários extremos, um agente mal alinhado ou comprometido pode produzir efeitos concretos e graves sobre sistemas reais [Kong et al. 2025].

Dependência de Contexto deve-se ao fato que a desempenho dos modelos é limitado pela janela de contexto, que restringe a quantidade de informação processável simultaneamente [de Lamo Castrillo et al. 2025]. Além disso, pequenas variações na redação de *prompts* podem levar a comportamentos distintos, o que torna a engenharia de *prompts* um elemento central na construção de sistemas mais estáveis [Bommasani et al. 2022]. Em redes, isso se torna ainda mais sensível quando a tarefa depende de documentação extensa, estado atualizado do ambiente e múltiplas restrições operacionais.

Essas limitações reforçam a necessidade de mecanismos de verificação automática, supervisão humana e implementação de *guardrails* antes de qualquer aplicação em redes de produção [Elkael et al. 2026].

Capacidades e Riscos Específicos em Tarefas de Rede

As limitações descritas anteriormente assumem formas concretas quando o domínio é a gerência de redes. Do lado das capacidades, os modelos demonstram desempenho consistente em tarefas bem delimitadas, como geração de trechos padronizados de configuração (e.g., interfaces, VLANs, ACLs e prefixos de roteamento) a partir de instruções em linguagem natural; *parsing* e sumarização de logs; tradução de intenção operacional para sintaxe de equipamentos quando exemplos do fabricante estão presentes no contexto; e explicação de mensagens de falha para operadores menos experientes [Chakraborty et al. 2024, Lira et al. 2024].

O risco, contudo, cresce em cenários mais sensíveis. Em topologias inéditas, combinações de equipamentos e protocolos ausentes dos dados de treinamento podem levar o modelo a extrapolar, gerando configurações sintaticamente válidas, porém semanticamente incorretas. Em ambientes multivendedor, a mistura de sintaxes e convenções específicas de fabricantes distintos aumenta a probabilidade de alucinações em campos delicados. Em comandos com efeito imediato e irreversível, como desligamento de interfaces, remoção de rotas ou alterações em ACLs em produção, o custo de uma decisão incorreta pode ser operacionalmente crítico [Boateng et al. 2024b]. Além disso, quando a tarefa exige raciocínio sobre estado global, o modelo corre o risco de decidir com base apenas no contexto textual fornecido, e não no estado real e atualizado da rede.

Essa assimetria define uma estratégia prudente de adoção: agentes podem atuar como primeira linha em tarefas rotineiras e bem delimitadas, mas com revisão humana ou verificação automática obrigatória antes da aplicação de qualquer configuração em produção [Lira et al. 2024]. Em outras palavras, a inteligência do agente deve ser combinada com controle operacional rigoroso.

Instrução em Linguagem Natural à Configuração Aplicada

Para ilustrar de forma integrada os conceitos discutidos, incluindo interpretação de intenção, planejamento, uso de memória, execução e verificação, apresentamos, a seguir, o fluxo completo de operação de um agente inteligente a partir de uma instrução em linguagem natural até a aplicação efetiva de uma configuração na rede.

Considere um operador que instrui o agente:

"Isole a VLAN 200 na interface eth2 do switch core-01 e confirme que o trunk para o switch dist-02 continua operacional."

1. Percepção e decomposição. A instrução chega como texto ao sistema. O modelo identifica duas subtarefas principais: configurar o isolamento da VLAN em uma interface de acesso e verificar o estado do *trunk* para outro equipamento. Esse passo corresponde à interpretação inicial da intenção e à decomposição do objetivo em ações menores.

2. Consulta à memória de longo prazo. O agente invoca mecanismos de recuperação para buscar documentação do fabricante do *core-01*, padrões operacionais internos e *playbooks* relacionados a VLANs. O conteúdo recuperado é inserido no contexto como suporte factual à tomada de decisão.

3. Geração da configuração. Com parâmetros de geração controlados, o modelo produz os comandos na sintaxe apropriada do equipamento. A saída idealmente não é texto livre, mas uma estrutura processável pelo executor, por exemplo em JSON:

```
{
  device: "core-01",
  commands:
  [
    "interface eth2",
    "switchport mode access",
    "switchport access vlan 200"
  ]
}
```

4. Verificação antes da aplicação. Antes da execução, um módulo verificador analisa a configuração produzida de acordo com as políticas definidas e das restrições do ambiente. Uma vez aprovada, a ação é enviada ao dispositivo por meio de mecanismos como SSH, NETCONF ou APIs equivalentes, e os resultados são coletados.

5. Observação e avaliação. O retorno da execução volta ao agente, que avalia se o estado obtido corresponde ao objetivo inicial. Caso o *trunk* para o *dist-02* permaneça operacional, a tarefa é concluída e registrada. Caso contrário, o agente formula uma ação corretiva, ajusta seu plano e itera.

Esse fluxo, que um operador humano poderia executar em vários minutos mediante consultas manuais à documentação, pode ser realizado em segundos, com rastreabilidade completa. Esse é um dos ganhos práticos mais imediatos da arquitetura de agentes aplicada a redes [Lira et al. 2024, Chakraborty et al. 2024].

Dessa forma, a combinação entre modelos de linguagem, memória, ferramentas e mecanismos de validação estabelece a base para a construção de sistemas capazes de atuar diretamente sobre a infraestrutura de rede. No entanto, a materialização desses conceitos exige uma arquitetura prática que integre decisão, execução e controle operacional. Essa transição entre fundamento conceitual e implementação será aprofundada nas seções seguintes.

3.3. Arquiteturas de Agentes com SLMs e LLMs

A construção de agentes inteligentes para redes pode seguir diferentes arquiteturas, cuja escolha depende diretamente de requisitos como latência, privacidade, custo operacional, capacidade de raciocínio e grau de autonomia desejado. Em ambientes de rede, essa decisão é particularmente sensível, pois o sistema precisa equilibrar precisão técnica, controle sobre dados sensíveis e tempo de resposta compatível com a operação. Não existe, portanto, uma arquitetura única e universalmente superior. Em vez disso, diferentes arranjos podem ser mais adequados conforme o contexto de implantação e o tipo de tarefa a ser executada.

3.3.1. Possíveis Abordagens

Uma primeira abordagem consiste no uso de um **LLM externo acoplado a uma aplicação local**, normalmente implementada em Python, que atua como orquestradora. Nessa arquitetura, a aplicação local recebe a solicitação do operador, organiza o contexto, classifica a intenção, monta o *prompt* e envia a requisição a um modelo robusto hospedado externamente por meio de API [Lira et al. 2024, Kong et al. 2025]. O sistema LLM-NetCFG é um exemplo representativo dessa estratégia, pois utiliza um módulo intermediário para gerenciar os *prompts*, estruturar entradas e apoiar a classificação de intenções antes do envio ao modelo [Lira et al. 2024]. A principal vantagem dessa abordagem está no acesso a modelos de alta capacidade sem a necessidade de manter infraestrutura local de grande porte. Em contrapartida, há dependência de conectividade, maior latência de ponta a ponta e exposição potencial de dados sensíveis da rede.

Uma segunda arquitetura é o **tool-calling interno**, na qual o modelo é treinado ou configurado para emitir chamadas de ferramentas em formatos estruturados, como JSON, em vez de apenas produzir texto livre [Kong et al. 2025, de Lamo Castrillo et al. 2025]. Nessa abordagem, o modelo decide quais funções, APIs ou mecanismos de consulta devem ser invocados, seja para obter dados em tempo real, seja para executar ações sobre a infraestrutura [de Lamo Castrillo et al. 2025, OpenAI 2025]. Em redes, isso inclui, por exemplo, consultar telemetria, recuperar documentação, acionar validadores, interagir com controladores SDN ou gerar comandos para execução controlada. Essa arquitetura aumenta a capacidade operacional do agente, pois aproxima a geração de linguagem da ação efetiva sobre o ambiente.

Uma terceira possibilidade é o **SLM embarcado localmente**, que consiste na execução de modelos compactos diretamente em hardware local, servidores privados ou dispositivos de borda. Modelos como Phi-4-mini, LFM2.5 e Llama-3.2-3B ilustram essa tendência [Zhou et al. 2024, Microsoft Research 2025, Liquid AI 2025]. Essa arquitetura é especialmente adequada para cenários em que baixa latência, confidencialidade e in-

dependência de conectividade são requisitos centrais [Zhou et al. 2024, Lira et al. 2024]. Em ambientes de redes corporativas, industriais ou de telecomunicações, a execução local reduz a exposição de informações operacionais e permite respostas mais rápidas em tarefas rotineiras, como geração de configurações, classificação de incidentes e validação estrutural de comandos.

Por fim, destacam-se as **arquiteturas híbridas**, nas quais diferentes modelos coexistem em um mesmo *pipeline*, com papéis distintos e complementares. Nesses casos, um componente roteador decompõe tarefas complexas e as distribui entre modelos especializados [Kong et al. 2025]. Modelos com maior capacidade de raciocínio podem ser utilizados para planejamento e análise de casos complexos, enquanto modelos menores e mais rápidos executam subtarefas específicas, como preenchimento de argumentos, chamadas de ferramentas e geração estruturada de saídas [Bandara et al. 2025, OpenAI 2025]. Essa abordagem permite equilibrar custo, latência e qualidade de raciocínio, tornando-se particularmente adequada para ambientes de rede nos quais coexistem tarefas simples e frequentes com situações raras de alta complexidade.

3.3.2. Arquiteturas Avançadas de Raciocínio e Eficiência Energética

Além da escolha da arquitetura geral do sistema, o desempenho de um agente depende também do padrão de raciocínio adotado para organizar o ciclo de decisão e ação. O *framework* ReAct [Yao et al. 2023] estabeleceu um paradigma influente ao propor a sequência Observação $\rightarrow$ Pensamento $\rightarrow$ Ação, permitindo que o modelo intercale raciocínio textual com interação sobre o ambiente. Essa formulação foi importante por tornar o processo mais interpretável e adaptativo. No entanto, em cenários de configuração de redes, nos quais uma sequência incompleta de passos pode produzir falhas em cascata, arquiteturas mais recentes vêm buscando superar suas limitações.

Entre essas abordagens, o **Pre-Act** [Rawat et al. 2025] introduz planejamento explícito antes da execução. Em vez de decidir incrementalmente a cada interação, o agente constrói inicialmente um plano mais completo e detalhado, refinando-o ao longo da execução. Essa característica é particularmente relevante em redes, onde a omissão de um passo intermediário pode comprometer a aplicação correta de VLANs, ACLs, rotas ou políticas distribuídas. Em avaliações, Pre-Act superou ReAct em 70% na métrica *Action Recall*, resultado importante para cenários em que a completude da sequência operacional é crítica. Outra proposta relevante é o **Structured Cognitive Loop (SCL/R-CCAM)** [Kim 2026], que organiza a cognição em cinco fases modulares: *Retrieval*, *Cognition*, *Control*, *Action* e *Memory*. Sua principal contribuição está na introdução de uma camada explícita de controle simbólico, aumentando explicabilidade e aderência a políticas. Em ambientes de produção, essa característica é especialmente valiosa, pois a rede exige não apenas decisões corretas, mas decisões auditáveis e compatíveis com restrições operacionais rígidas.

Já o **Modular Agentic Planner (MAP)** [Webb et al. 2025] propõe decompor o planejamento em módulos especializados, inspirados no funcionamento do córtex pré-frontal humano. Nesse arranjo, diferentes módulos assumem tarefas como monitoramento de erros, proposição de ação, predição de estados e avaliação de resultados. Em estudos comparativos, o MAP superou abordagens como *Chain-of-Thought*, *Multi-Agent Debate*

e *Tree-of-Thought* em diferentes domínios de planejamento. Em redes, esse tipo de modularidade é promissor porque aproxima o raciocínio do agente da própria natureza distribuída e multifásica da operação de infraestrutura. A Tabela 3.3 apresenta uma comparação de arquiteturas avançadas de raciocínio de agentes.

Tabela 3.3. Comparação de arquiteturas avançadas de raciocínio para agentes

Arquitetura	**Abordagem Principal**	**Vantagem-Chave**
ReAct [Yao et al. 2023]	Thought → Action → Observation	Interpretabilidade e adaptabilidade
Pre-Act [Rawat et al. 2025]	Plano completo antes da execução	+70% Action Recall vs ReAct
SCL (R-CCAM) [Kim 2026]	5 fases modulares com controle simbólico	Zero violações de política
MAP [Webb et al. 2025]	Módulos inspirados no PFC humano	Melhor generalização entre tarefas

A dimensão energética também merece atenção especial em sistemas de agentes aplicados a redes. Medições da plataforma ML.ENERGY [ML.ENERGY 2024] indicam que modelos de grande porte, como o LLaMA-65B, podem consumir aproximadamente 3 a 4 J/token, enquanto SLMs operam na faixa de 0,6 mJ/token, uma diferença de até 5.800 vezes. Em experimentos com robótica móvel, observou-se que SLMs foram capazes de restaurar a latência do loop sensório-motor a níveis aceitáveis em tempo real, enquanto LLMs maiores introduziram atrasos proibitivos. Em redes, especialmente em cenários com restrições de latência sub-segundo, esse aspecto deixa de ser apenas econômico e passa a ser operacionalmente decisivo.

3.3.3. Adaptação de SLMs para Cenários de Agentes: *Fine-Tuning* e Quantização

A viabilidade dos SLMs em *pipelines* agênticos depende menos do número absoluto de parâmetros e mais da qualidade do processo de adaptação ao domínio. Modelos na faixa de 1 a 7 bilhões de parâmetros, quando ajustados sobre tarefas de *tool-calling*, geração estruturada ou configuração de redes, frequentemente superam modelos ordens de grandeza maiores que operam apenas via *prompting* genérico [Jhandi et al. 2026, Sharma and Mehta 2025]. Esse resultado é particularmente importante para o domínio de redes, no qual grande parte das tarefas é recorrente, estruturada e dependente de formatos específicos de saída.

O modelo OPT-350M, ajustado sobre o conjunto ToolBench, pode atingir 77,55% de taxa de sucesso em *benchmarks* de chamada de ferramentas, contra 26% do ChatGPT com *chain-of-thought* [Jhandi et al. 2026]. Nesse caso, o treinamento completo levou menos de 15 minutos em hardware de consumidor, indicando que, em tarefas delimitadas, a especialização de um modelo pequeno pode ser mais eficaz do que o uso indiscriminado de modelos muito maiores. A técnica dominante nesse processo é o **QLoRA** [Dettmers et al. 2023], que combina um modelo base quantizado em 4 bits, no formato NF4, com adaptadores de baixo ranque mantidos em precisão plena. Essa estratégia re-

duz drasticamente o consumo de memória, sem perda estatisticamente significativa de desempenho. Na fase de implantação, essa eficiência pode ser ampliada por formatos de quantização pós-treinamento. O **GPTQ** utiliza informação de segunda ordem para minimizar o erro de reconstrução dos pesos em GPU, enquanto o **GGUF**, formato nativo do *llama.cpp* e do Ollama, viabiliza inferência eficiente em CPU, com consumo de memória compatível com servidores de borda. Em avaliações com Llama 3.2-1B submetido a QLoRA seguido de quantização GPTQ 4 bits, a acurácia foi preservada em 0,99 em relação ao modelo de referência em precisão plena [Sharma and Mehta 2025]. Em termos práticos, isso sugere que um SLM executado localmente em um controlador de rede pode desempenhar tarefas de geração de configuração com qualidade comparável à de modelos muito maiores, desde que o domínio de ajuste seja aderente ao ambiente operacional.

O projeto **TinyAgent** [Erdogan et al. 2024] reforça essa hipótese ao mostrar que modelos de 1,1B e 7B parâmetros, ajustados com mecanismos de recuperação de ferramentas para compressão de *prompt*, puderam igualar ou superar o GPT-4-Turbo em benchmarks de chamada de funções, com execução inteiramente local. Para redes industriais, infraestruturas críticas e ambientes com exigências rígidas de confidencialidade, essa característica é particularmente relevante, pois elimina a necessidade de expor topologias, configurações e políticas de roteamento a APIs externas.

Componentes Arquiteturais

A construção de um agente robusto exige a integração coordenada de diferentes módulos funcionais. Em vez de tratar o agente como um único modelo, arquiteturas modernas o definem como um sistema composto por elementos especializados, cada um responsável por uma parte do ciclo operacional.

O **Agent Model** constitui o núcleo cognitivo do sistema, podendo ser um LLM ou um SLM. Esse componente é responsável pela compreensão da linguagem, pelo raciocínio lógico, pela interpretação da intenção e pela formulação do plano de ação [Kong et al. 2025, de Lamo Castrillo et al. 2025].

O **Tool Executor** converte as decisões abstratas do modelo em ações concretas sobre a infraestrutura, como execução de comandos, chamadas de API ou interação com dispositivos e serviços externos [Kong et al. 2025, de Lamo Castrillo et al. 2025]. Em redes, esse componente é central porque estabelece a ponte entre geração semântica e operação técnica real.

O **Verifier** atua como um auditor de segurança, sintaxe e conformidade, validando as configurações geradas antes da aplicação na rede. Esse papel pode ser exercido por regras estáticas, validadores específicos ou ferramentas como Batfish, no caso de análise de políticas e ACLs [Lira et al. 2024].

O **Router** coordena o fluxo entre modelos, agentes ou ferramentas, decidindo qual recurso deve ser utilizado em cada subtarefa [Kong et al. 2025, OpenAI 2025]. Em arquiteturas híbridas, ele é o elemento que viabiliza o padrão SLM-default, LLM-fallback.

A **Memory** é essencial para manter continuidade operacional. Em geral, combina memória de curto prazo, associada ao contexto imediato da interação, com memória de longo prazo, baseada em histórico de interações, estados anteriores, documentos e deci-

sões passadas [Kong et al. 2025].

O **RAG (*Retrieval-Augmented Generation*)** amplia a base factual do agente ao permitir consulta a documentos externos, como manuais técnicos, logs, playbooks e históricos operacionais, reduzindo alucinações e aumentando a precisão da resposta [Kong et al. 2025, de Lamo Castrillo et al. 2025].

Por fim, os mecanismos de **Logging e Auditoria** registram comunicações, decisões e ações executadas, garantindo rastreabilidade, explicabilidade e possibilidade de revisão posterior, características indispensáveis em ambientes críticos [Kong et al. 2025].

3.3.3.1. Arquiteturas Heterogêneas e Roteamento entre SLMs e LLMs

A oposição rígida entre SLMs e LLMs vem perdendo sentido nas arquiteturas de produção mais recentes. Em vez de escolher um único modelo para todo o sistema, arquiteturas heterogêneas distribuem diferentes funções entre modelos distintos, com um componente de roteamento responsável por alocar cada subtarefa ao recurso mais eficiente para aquele contexto [Ye et al. 2025, Yue et al. 2025]. Essa mudança reflete uma compreensão mais realista dos sistemas agênticos: tarefas diferentes impõem exigências diferentes, e raramente um único modelo atende de forma ótima a toda a cadeia de processamento.

O benchmark X-MAS [Ye et al. 2025], que avaliou 27 modelos em cinco domínios por meio de 1,7 milhão de execuções, mostrou que configurações heterogêneas podem superar arquiteturas homogêneas em até 8,4% em raciocínio matemático e em até 47% nos problemas mais complexos da competição AIME. Embora esses números não sejam específicos de redes, eles reforçam o princípio de que a seleção ótima de modelo depende da natureza da subtarefa, e não apenas de uma métrica global de capacidade.

O **MasRouter** [Yue et al. 2025] operacionaliza esse princípio ao adotar uma rede controladora em cascata que decide o modo de colaboração, aloca papéis e roteia cada requisição ao modelo mais apropriado. Em benchmarks de geração de código, o sistema obteve melhora de 1,8 vezes sobre o estado da arte, com redução de custo de 52 vezes em relação ao uso exclusivo de LLM no HumanEval. Dessa lógica emerge o padrão *SLM-default, LLM-fallback*, no qual tarefas rotineiras são encaminhadas a SLMs locais e apenas casos ambíguos, raros ou semanticamente incertos são escalados a LLMs mais robustos [NVIDIA Research 2025]. A Tabela 3.4 sumariza diferentes padrões de implantação de modelos em *pipelines* agênticos para gerência de redes.

No contexto de gerenciamento de redes, esse padrão é bastante aderente à natureza dual das operações. A maior parte das tarefas é repetitiva e bem delimitada, como geração de configurações, verificação de sintaxe, validação de ACLs e classificação de incidentes. Apenas uma fração menor exige raciocínio sobre cenários inéditos, múltiplos sintomas distribuídos ou eventos fora da distribuição de treinamento [Long et al. 2025]. Usar LLMs em todas as etapas gera sobredimensionamento computacional e financeiro; usar apenas SLMs pode limitar a capacidade de resposta a exceções complexas. O roteamento adaptativo resolve essa tensão de forma sistemática [Sharma and Mehta 2025, Kong et al. 2025].

Tabela 3.4. Padrões de implantação de modelos em *pipelines* agênticos para gerência de redes

Padrão	Lógica de roteamento	Caso de uso em redes	Custo relativo
SLM exclusivo	Todas as tarefas no modelo local	Configs rotineiras, baixa latência exigida	Muito baixo
LLM exclusivo	Todas as tarefas via API externa	Diagnóstico complexo sem restrição de custo	Alto
SLM-default/ LLM-fallback	SLM para tarefas conhecidas; LLM para anomalias	Operações diárias com escalonamento pontual	Baixo a médio
Heterogêneo com router	Roteamento dinâmico por complexidade e domínio	Sistemas multiagente autônomos de grande escala	Variável

3.3.4. Comparação com Automação Tradicional

Em relação às abordagens clássicas de automação, os agentes baseados em IA introduzem uma camada deliberativa ausente em ferramentas puramente declarativas. Soluções como Ansible, NETCONF e gNMI continuam fundamentais, mas operam a partir de entradas estruturadas, esquemas previamente definidos e fluxos explicitamente modelados [Huang et al. 2023, Boateng et al. 2024b]. Essa característica lhes confere determinismo e auditabilidade, mas também reduz sua flexibilidade diante de cenários imprevistos, dados não estruturados ou objetivos expressos em linguagem natural.

Ferramentas como o **Ansible** executam scripts e playbooks predefinidos, sendo extremamente úteis em tarefas repetitivas e controladas. No entanto, sua lógica não contempla, por si só, interpretação de intenção, replanejamento dinâmico ou diagnóstico semântico de falhas. Já um **agente deliberativo** utiliza um modelo de linguagem para interpretar a solicitação do operador, planejar as etapas de execução e adaptar seu comportamento com base no retorno do ambiente [Lira et al. 2024, Elkael et al. 2026]. Isso permite, por exemplo, correlacionar mensagens de erro, ajustar um plano intermediário e até corrigir trechos de automação tradicional que tenham falhado [Boateng et al. 2024b].

A comparação torna-se ainda mais clara quando analisamos protocolos específicos. O **NETCONF**, definido pela RFC 6241, oferece um protocolo de gerenciamento baseado em XML sobre SSH, com operações transacionais como `edit-config` e `get-config`, o que garante atomicidade e capacidade de rollback. Sua principal força está na consistência operacional em ambientes multivendedor. Sua limitação, contudo, está na rigidez: o operador precisa conhecer previamente o esquema YANG e a estrutura dos dados envolvidos.

O **gNMI**, por sua vez, adota transporte gRPC e oferece suporte nativo a telemetria em tempo real, tornando-se especialmente útil em ambientes modernos de coleta frequente e observabilidade contínua. Ainda assim, o operador continua precisando conhecer os caminhos exatos dos dados que deseja consultar ou modificar.

O agente inteligente não substitui essas ferramentas, mas opera em uma ca-

mada superior. Ele interpreta a intenção em linguagem natural, decide quais operações NETCONF ou gNMI são necessárias, gera os payloads apropriados, envia as requisições e analisa os resultados devolvidos pelo dispositivo. Quando há falha, o agente pode reformular a ação sem intervenção humana imediata. Em outras palavras, a automação tradicional fornece a interface determinística e auditável; o agente fornece a inteligência adaptativa que a operacionaliza [Huang et al. 2023, Lira et al. 2024].

3.3.5. Infraestrutura de Execução

A viabilidade operacional de agentes inteligentes depende também da infraestrutura utilizada para inferência e execução. A escolha do hardware afeta latência, custo, consumo energético e escalabilidade do sistema.

Em termos gerais, tarefas como recuperação de informação, indexação e pré-processamento podem ser executadas adequadamente em CPU. Já a inferência de modelos maiores tende a exigir GPUs de alto desempenho para atingir tempos de resposta compatíveis com uso interativo ou automação em tempo quase real [Khattab and Zaharia 2020]. Esse aspecto é particularmente relevante em sistemas de agentes, nos quais uma única tarefa pode envolver múltiplas iterações entre modelo, executor e verificador.

LLMs de grande porte apresentam maior custo por token, maior latência e maior demanda computacional. Em contraste, SLMs ajustados e quantizados, especialmente quando combinados com técnicas como LoRA e GGUF, podem oferecer maior vazão de tokens e menor custo operacional [Huang et al. 2023]. Em redes móveis avançadas, infraestruturas 6G ou aplicações críticas com requisitos estritos de tempo de resposta, a latência imposta por modelos remotos ou excessivamente grandes pode tornar-se inviável [Zhou et al. 2024].

Em termos práticos, a escolha da plataforma precisa considerar não apenas desempenho bruto, mas adequação ao contexto da rede. Para tarefas rotineiras, controladas e frequentes, a execução local em CPU ou GPU modesta pode ser suficiente. Para diagnósticos mais complexos ou cenários com forte exigência de raciocínio, pode ser necessário recorrer a infraestrutura mais robusta. A Tabela 3.5 faz uma comparação de plataformas para inferências de modelos.

Tabela 3.5. Comparação de plataformas para inferência de modelos

Plataforma	Tokens/s	Latência	Custo	Maturidade
CPU	5–25	Alta	Baixo	Alta
NVIDIA CUDA	40–300+	Baixa	Alto	Muito Alta
AMD ROCm	60–200	Baixa	Médio	Média
Apple Metal	30–120	Baixa	Médio	Alta (ecossistema Apple)

3.3.6. Classificação por Ambiente

Além do *hardware*, a implantação dos modelos pode ocorrer em diferentes camadas da infraestrutura, cada uma com vantagens e limitações próprias.

No modelo **Cloud**, a inferência é realizada em provedores externos, com acesso a recursos computacionais praticamente ilimitados para treinamento e raciocínio pesado.

Essa abordagem facilita o uso de modelos de ponta, mas aumenta latência e amplia riscos associados à privacidade dos dados [Lira et al. 2024].

Na modalidade **Local**, os modelos são executados dentro da própria rede privada do operador. Isso garante maior controle sobre segurança e confidencialidade, além de reduzir a dependência de conectividade externa [Lira et al. 2024]. Em redes corporativas e ambientes críticos, essa costuma ser uma opção particularmente atraente.

Já na implantação em **Edge**, o modelo é posicionado o mais próximo possível do ponto de operação, como servidores de borda ou estações rádio base. Essa estratégia é indicada para tarefas sensíveis à latência e a decisões em tempo real, embora seja limitada pela capacidade de hardware disponível no dispositivo [Zhou et al. 2024].

Dessa forma, a classificação por ambiente complementa a escolha arquitetural e a escolha de hardware. Em agentes para redes, a decisão final raramente depende de um único fator. Ela resulta da combinação entre requisitos de privacidade, tempo de resposta, custo, criticidade operacional e complexidade do raciocínio esperado.

Como consequência, as arquiteturas mais promissoras para o domínio de redes tendem a combinar execução local ou de borda para tarefas frequentes e sensíveis, com escalonamento seletivo para modelos mais robustos em nuvem quando a complexidade da tarefa justificar esse custo adicional. Essa visão prepara o terreno para a próxima seção, na qual o foco passa do desenho arquitetural para o paradigma operacional da IA Agentic.

3.4. IA Agêntica: Paradigma Moderno para a Construção de Agentes Inteligentes

A IA Agêntica representa uma transição fundamental dos chatbots reativos para sistemas autônomos capazes de raciocinar, planejar e executar tarefas em múltiplas etapas em infraestruturas complexas [Bandara et al. 2025]. Diferentemente do uso tradicional de LLMs por meio de *prompts* únicos, os fluxos agênticos utilizam o modelo de linguagem como um núcleo cognitivo que orquestra ferramentas, gerencia memória e interage dinamicamente com o ambiente [Du et al. 2026, Bandara et al. 2025].

Ciclo deliberativo

O modelo conceitual que orienta a construção desses agentes baseia-se em um ciclo fechado de percepção e ação, permitindo que a IA não apenas gere texto, mas atue como um colaborador inteligente [de Lamo Castrillo et al. 2025].

Perceber (*Perceive*): o agente inicia sua interação capturando dados do ambiente [de Lamo Castrillo et al. 2025]. Em redes, isso inclui o monitoramento de condições de tráfego, logs e indicadores-chave de desempenho (KPIs) [Elkael et al. 2026]. O sistema de percepção transforma esses perceptos brutos em representações significativas que o LLM pode processar, como o estado atual de uma topologia ou uma falha de conectividade detectada [de Lamo Castrillo et al. 2025].

Planejar (*Plan*): com base na percepção e na intenção do operador, por exemplo, "priorizar tráfego de vídeo na Região A", o agente utiliza sua capacidade de raciocínio para decompor o objetivo de alto nível em subtarefas gerenciáveis [Elkael et al. 2026,

de Lamo Castrillo et al. 2025]. O planejamento pode ser hierárquico, distribuindo a execução entre diferentes escalas de tempo, de milissegundos a minutos, e entre diferentes domínios da rede, como núcleo, borda ou RAN [Elkael et al. 2026].

Agir (*Act*): nessa fase, o agente traduz decisões abstratas em operações concretas no mundo digital [de Lamo Castrillo et al. 2025]. O módulo de ação interage com o ambiente por meio de ferramentas externas, como chamadas de API, execução de scripts de configuração ou ajustes de parâmetros em controladores SDN [de Lamo Castrillo et al. 2025, Kong et al. 2025].

Avaliar (*Evaluate/Reflect*): após a execução, o agente entra em um estágio de reflexão e autoavaliação [de Lamo Castrillo et al. 2025]. Nessa etapa, ele analisa os resultados obtidos em relação à intenção original, no contexto de *intent assurance*, para identificar erros de sintaxe ou desvios de configuração, como *intent drift* [de Lamo Castrillo et al. 2025, Elkael et al. 2026]. Se detectar falhas, o agente pode corrigir proativamente seu plano e tentar uma nova abordagem sem intervenção humana imediata [de Lamo Castrillo et al. 2025, OpenAI 2025].

3.4.1. Capacidades Modernas

Os agentes inteligentes modernos superam o processamento estático de texto ao incorporar múltiplas capacidades complementares:

Integração com ferramentas: uso de calculadoras, analisadores de tráfego, simuladores e APIs para realizar operações que excedem as capacidades nativas do LLM [Kong et al. 2025].

Bases de conhecimento via RAG: a *Retrieval-Augmented Generation* (RAG) permite que o agente consulte manuais técnicos, históricos de rede e bases documentais em tempo real, reduzindo alucinações e aumentando a fundamentação factual das respostas [Kong et al. 2025].

Memória de curto e longo prazo: o agente preserva o histórico da sessão para manter coerência interacional e também retém experiências passadas, possibilitando aprendizado a partir de erros e decisões anteriores [Derouiche et al. 2025].

3.4.2. Ecossistema de *Frameworks*

O desenvolvimento desses agentes é facilitado por *frameworks* que padronizam a orquestração, a memória e a comunicação entre componentes:

LangChain Agents: oferece abstrações modulares para planejamento, memória e invocação de ferramentas [Du et al. 2026, Kong et al. 2025].

AutoGen: enfatiza a colaboração multiagente, permitindo que diferentes perfis de IA, como planejadores e verificadores, interajam para resolver problemas complexos [Du et al. 2026, Derouiche et al. 2025].

CrewAI: especializado em processos de trabalho orientados por papéis, permitindo a criação de equipes de agentes com responsabilidades específicas [Derouiche et al. 2025].

OpenAI Swarm e OpenWebUI Tools: representam abordagens emergentes vol-

tadas à orquestração leve de agentes e à integração de ferramentas em interfaces modernas.

Esse ciclo deliberativo transforma o gerenciamento de redes em um sistema de auto-operação, no qual a inteligência evolui continuamente a partir dos dados operacionais e das interações com o ambiente [Elkael et al. 2026].

Comparativo de *frameworks*, protocolos de interoperabilidade e *spec-driven development*

A escolha do *framework* influencia diretamente as características operacionais de todo o sistema de agentes, como indicado na Tabela 3.6.

Tabela 3.6. Comparativo de frameworks de agentes (2025–2026)

Framework	Força principal	Melhor aplicação
LangGraph [LangChain 2025]	*Workflows* com estado e grafos cíclicos	*Workflows* complexos com múltiplos pontos de decisão
CrewAI	Coordenação multiagente orientada por papéis	Prototipagem rápida de sistemas multiagente
AutoGen	Arquitetura orientada a eventos entre agentes	Pesquisa e *workflows* com supervisão humana
LlamaIndex	Indexação e recuperação avançada	Aplicações com forte dependência de RAG
Semantic Kernel	Integração entre LLMs e linguagens corporativas	Ambientes Microsoft e Azure

Model Context Protocol (MCP) [Anthropic 2025]: o MCP consolidou-se como um padrão de fato para integração entre agentes e ferramentas. Baseia-se em mensagens JSON-RPC 2.0 para estabelecer comunicação entre três entidades: *hosts*, que iniciam as conexões; *clients*, que funcionam como conectores internos; e *servers*, que disponibilizam contexto e capacidades. A evolução do protocolo ampliou sua aplicabilidade, inclusive em padrões confiáveis do tipo *call-now/fetch-later*. Entre seus riscos específicos, destacam-se injeção de *prompt* por descrições de ferramentas e escalada de privilégio entre sistemas [Errico et al. 2025].

Agent-to-Agent Protocol (A2A) [Google 2025]: o A2A foi concebido para permitir que agentes implementados em diferentes *frameworks* descubram uns aos outros, autentiquem-se e interajam independentemente da tecnologia subjacente. Seus componentes incluem *Agent Cards*, para descoberta padronizada de capacidades, *Task Management*, para controle do ciclo de vida das tarefas, e mecanismos de segurança corporativa, como OAuth 2.0, chaves de API e mTLS. MCP e A2A são complementares: o primeiro conecta agentes a ferramentas; o segundo organiza a comunicação entre agentes.

***Spec-Driven Development* (SDD)**: quando o custo marginal de geração de código se aproxima de zero, o principal ativo cognitivo deixa de ser o código em si e passa a ser a especificação formal. No SDD, o agente recebe uma especificação estruturada, como

um *schema* JSON, contrato OpenAPI ou grafo de estados, e gera a implementação como artefato derivado, iterando até que verificadores automáticos atestem conformidade. Em redes, isso significa que a intenção do operador, expressa formalmente, torna-se o documento canônico, enquanto a configuração do dispositivo passa a ser apenas sua projeção executável.

3.4.3. *Feedback Loop* Iterativo

A transição de modelos de linguagem para agentes inteligentes ocorre por meio de um sistema de malha fechada (*closed-loop system*), que estabelece uma cadeia contínua de percepção, decisão, ação e realimentação [Kong et al. 2025]. Diferentemente do paradigma de *prompt* único, o agente opera em um ciclo deliberativo que lhe permite executar tarefas complexas e corrigir erros de forma proativa. O fluxo desse ciclo pode ser descrito em quatro etapas principais:

1. **O LLM gera o comando**: o modelo atua como núcleo cognitivo do sistema [de Lamo Castrillo et al. 2025]. Utilizando estratégias como ReAct, intercala raciocínio verbal com a produção de ações específicas [Yao et al. 2023]. Nessa etapa, decide qual ferramenta ou API deve ser invocada e emite comandos estruturados, geralmente em JSON, descrevendo os parâmetros da ação [Kong et al. 2025].
2. **O executor realiza a ação**: as decisões abstratas produzidas pelo LLM são encaminhadas a um módulo executor ou orquestrador. Esse componente traduz o comando em operações reais, como chamadas de API, execução de scripts ou consultas a bases de dados [Lira et al. 2024, de Lamo Castrillo et al. 2025]. Em sistemas como o LLM-NetCFG, o orquestrador garante que a configuração seja enviada corretamente aos dispositivos ou simuladores [Lira et al. 2024].
3. **O resultado retorna ao agente**: após a execução, o ambiente ou a ferramenta externa devolve um resultado, correspondente à fase de observação no paradigma ReAct [Yao et al. 2023]. Esse retorno pode assumir a forma de sucesso operacional, dados de telemetria ou mensagens de erro. No Toolformer, por exemplo, o modelo aprende a utilizar ferramentas avaliando se a resposta obtida é útil para reduzir a incerteza da tarefa [Schick et al. 2023].
4. **O modelo decide continuar ou finalizar**: com base no retorno recebido, o agente reavalia o estado da tarefa. Por meio de mecanismos de autorrefinamento, como o *Self-Refine*, o modelo analisa se o objetivo foi alcançado ou se é necessário ajustar a estratégia [Madaan et al. 2023, Du et al. 2026]. O ciclo prossegue até que uma condição de saída seja satisfeita, como conclusão da tarefa, ocorrência de erro fatal ou atingimento do número máximo de tentativas [OpenAI 2025].

Esse *loop* iterativo é central para o NetOps 2.0, pois permite que o sistema trate falhas de configuração de maneira autônoma. Se um verificador detectar inconsistências, o agente recebe esse retorno e produz uma nova configuração corrigida no ciclo seguinte [Lira et al. 2024].

3.4.4. Planejamento Multietapas

O planejamento multietapas é uma das características que distinguem um agente inteligente de um modelo de linguagem estático. Ele permite decompor objetivos complexos em sequências de tarefas menores e manejáveis, utilizando o LLM como núcleo cognitivo para orquestrar a execução e lidar com incertezas [Zhou et al. 2024, de Lamo Castrillo et al. 2025].

Cadeias de decisão: agentes modernos empregam estratégias de decomposição para navegar em problemas complexos. Essa decomposição pode ocorrer de forma sequencial, quando todas as submetas são definidas antes da execução, ou de forma intercalada, com planejamento dinâmico a cada passo, como ocorre em abordagens como *Chain-of-Thought* e ReAct [de Lamo Castrillo et al. 2025, Huang et al. 2023]. Em redes, isso permite ao agente analisar topologia, estado do hardware e configurações de software antes de emitir comandos finais [Huang et al. 2023].

***Rollback* e refinamento de planos**: a autonomia confiável exige mecanismos para lidar com falhas. O agente precisa ser capaz de realizar autorreflexão, retroceder e reformular o plano sempre que um comando falha ou um verificador detecta inconsistências, como erros de sintaxe em ACLs [Zhou et al. 2024, de Lamo Castrillo et al. 2025, Lira et al. 2024]. Estratégias de reflexão antecipatória permitem, inclusive, prever falhas antes da execução efetiva [de Lamo Castrillo et al. 2025].

Diagnóstico automático: em redes de larga escala, LLMs e agentes transformam o *troubleshooting* de uma atividade manual em um processo assistido por IA [Huang et al. 2023, Boateng et al. 2024b]. O agente pode realizar análise de causa raiz (*Root Cause Analysis*, RCA) a partir de logs não estruturados e telemetria em tempo real, identificando gargalos, falhas de conectividade e degradações de desempenho [Boateng et al. 2024b, Bandara et al. 2025]. Com base nisso, gera relatórios e propõe ações corretivas, contribuindo para o objetivo de auto-manutenção do NetOps 2.0 [Huang et al. 2023, Lira et al. 2024].

3.4.5. Modelos Auxiliares

Os sistemas modernos de IA Agentic deixaram de ser modelos isolados para tornar-se arquiteturas modulares, nas quais modelos auxiliares desempenham funções críticas na mitigação de limitações como alucinações, ausência de memória persistente e falhas de roteamento [Bandara et al. 2025, de Lamo Castrillo et al. 2025].

Verifier model: atua como auditor de qualidade e segurança, avaliando as saídas geradas pelo modelo principal antes da execução final. No contexto de redes, sistemas como o LLM-NetCFG empregam módulos verificadores para checar erros de sintaxe e inconsistências em configurações geradas [Lira et al. 2024]. Esse verificador pode ser implementado por heurísticas, por outro LLM configurado como crítico ou por ferramentas externas, como o Batfish [Du et al. 2026, Lira et al. 2024].

Memory model: gerencia a retenção de informações que não estão contidas nos pesos do modelo. Permite que o agente preserve contexto em interações de múltiplos turnos e aprenda com experiências passadas [de Lamo Castrillo et al. 2025, Kong et al. 2025]. Em geral, distingue-se memória de curto prazo, associada ao contexto imediato, e memó-

ria de longo prazo, baseada em armazenamento externo, como bancos de dados vetoriais. Modelos mais avançados empregam memórias semântica, episódica e procedimental [Chowa et al. 2026, Derouiche et al. 2025].

Reranker: utilizado sobretudo em sistemas RAG para aumentar a precisão da recuperação de informação. Após uma busca inicial selecionar documentos candidatos, o *reranker* utiliza mecanismos mais refinados, como atenção cruzada, para reordenar os resultados e encaminhar ao LLM apenas as passagens mais relevantes [Karpukhin et al. 2020, Du et al. 2026].

Router: atua como roteador do fluxo de trabalho, delegando subtarefas a especialistas, ferramentas ou modelos específicos [de Lamo Castrillo et al. 2025]. Em arquiteturas *Mixture-of-Experts* (MoE) e em sistemas multiagente, esse componente decide qual recurso é mais adequado a cada demanda, otimizando custo e tempo de resposta [Fedus et al. 2022, Kong et al. 2025].

Esses modelos auxiliares tornam a arquitetura mais modular, auditável e controlável, características indispensáveis para a automação inteligente em ambientes críticos de telecomunicações [Bandara et al. 2025, Elkael et al. 2026].

3.5. Engenharia de *Prompt* para Agentes Aplicados a Redes

A estratégias de Engenharia de *Prompt* fundamentais para o desenvolvimento de agentes inteligentes voltados à automação de redes, com foco na criação de instruções robustas, seguras e operacionalmente confiáveis.

3.5.1. Papel do *System Prompt*

O *system prompt*, ou instrução de sistema, funciona como o "DNA" operacional do agente, definindo contexto, tarefa, restrições e diretrizes comportamentais que orientarão o Modelo de Linguagem de Grande Escala (LLM) [Lira et al. 2024, OpenAI 2025]. Diferentemente de um *prompt* de usuário comum, ele estabelece as bases para que o agente atue de forma confiável, consistente e alinhada aos objetivos do operador de rede [OpenAI 2025].

Definição da identidade do agente: A atribuição de uma persona ou identidade é um dos primeiros passos para condicionar o comportamento do modelo e delimitar sua política de atuação [Chowa et al. 2026]. No contexto de redes, o sistema LLM-NetCFG, por exemplo, define explicitamente o papel do agente como um "administrador de rede responsável por criar arquivos de configuração de dispositivos" [Lira et al. 2024]. Essa definição ajuda o modelo a adotar o tom técnico apropriado e a priorizar o conhecimento de domínio necessário, como a sintaxe de sistemas operacionais de rede, a exemplo do Cisco IOS [Lira et al. 2024].

Objetivos e escopo: As instruções de sistema devem delimitar com clareza o que o agente deve realizar e como deve decompor intenções de alto nível em passos técnicos [Lira et al. 2024, OpenAI 2025]. Recomenda-se instruir o agente a dividir tarefas complexas em subetapas menores, reduzindo ambiguidades e aumentando a precisão em configurações como VLANs, roteamento e ACLs [OpenAI 2025, de Lamo Castrillo et al. 2025]. O escopo também deve ser explicitamente restrito, de modo que o agente recuse consul-

tas fora de sua competência, por exemplo, perguntas sobre clima quando sua função é gerenciar roteadores [OpenAI 2025]. Além disso, deve-se orientar o modelo a basear-se estritamente em documentos técnicos e manuais fornecidos, a fim de reduzir alucinações [OpenAI 2025].

Restrições operacionais: Para garantir a segurança da infraestrutura, o *system prompt* deve impor limites rígidos de atuação. Isso inclui políticas de segurança, como a proibição de determinadas configurações de trânsito [Zhou et al. 2024], a exigência de verificação antes de qualquer aplicação real [Lira et al. 2024] e a limitação de ações irreversíveis sem supervisão humana. Uma prática comum consiste em impor restrições de saída, como a instrução: "Você não tem permissão para fornecer explicações; responda apenas com os arquivos de configuração" [Lira et al. 2024]. Isso evita ruído textual e garante que o executor receba apenas dados processáveis [Lira et al. 2024, Bandara et al. 2025]. Outras restrições podem incluir o uso de separadores especiais para isolar configurações de diferentes dispositivos em uma única resposta [Lira et al. 2024].

Formato obrigatório de saída: A comunicação determinística entre o agente e o executor depende do uso de formatos estruturados. Nesse contexto, o JSON (*JavaScript Object Notation*) é amplamente recomendado e, em muitos casos, obrigatório [Lira et al. 2024]. O JSON permite que o agente produza objetos bem formados contendo o nome da função e os argumentos técnicos necessários, como IP da interface, ID da VLAN ou métrica de rota, facilitando o *parsing* automático pelo sistema de automação e reduzindo erros de sintaxe [Kong et al. 2025]. Essa abordagem de comunicação de baixa contagem de tokens (*low-token communication*) aumenta a eficiência da transmissão e simplifica a auditoria do plano antes da execução [Kong et al. 2025, Bandara et al. 2025].

Ao consolidar esses elementos, o *system prompt* transforma um modelo de linguagem genérico em um agente deliberativo de rede, capaz de operar dentro de margens de segurança aceitáveis para ambientes de produção [OpenAI 2025, Lira et al. 2024].

3.5.2. Estrutura Recomendada

Para que um agente de rede opere de maneira determinística e segura, o *system prompt* deve seguir uma estrutura rigorosa, definindo não apenas o que o modelo deve fazer, mas também como ele deve se comportar diante de falhas, ambiguidades e restrições. Com base na literatura analisada, uma estrutura recomendada pode ser organizada em quatro pilares fundamentais:

Objetivo: define a meta principal do agente e orienta a decomposição de tarefas complexas em etapas executáveis [Chowa et al. 2026]. Esse objetivo deve alinhar as intenções de alto nível do operador (*business intents*) com configurações técnicas precisas (*resource intents*) [Lira et al. 2024]. Além disso, é recomendável instruir o agente a dividir tarefas densas em subetapas menores para minimizar ambiguidades [OpenAI 2025].

Regras de segurança: funcionam como *guardrails*, mitigando riscos de privacidade, segurança e conformidade operacional [OpenAI 2025]. Essas regras podem incluir mecanismos para detectar tentativas de *jailbreak*, injeção de *prompts* ou extração indevida de instruções internas [OpenAI 2025]. No plano operacional, devem proibir ações irreversíveis ou de alto impacto sem supervisão humana e exigir, por exemplo, respostas

exclusivamente em JSON para evitar ruído no executor [Lira et al. 2024, OpenAI 2025].

Política de execução: define o fluxo operacional do agente, determinando quando recorrer ao raciocínio do LLM e quando utilizar ferramentas externas. Recomenda-se reservar ferramentas determinísticas para operações críticas de infraestrutura, como escritas em banco de dados, *commits* ou alterações de configuração, deixando o LLM concentrado em tarefas que exijam interpretação semântica e planejamento [Bandara et al. 2025]. Essa política também pode explicitar o uso de protocolos como o MCP para descoberta e invocação dinâmica de funções [Elkael et al. 2026], bem como estratégias como ReAct para intercalar raciocínio, ação e observação [Yao et al. 2023].

Critérios de parada: todo ciclo de execução precisa de condições de saída bem definidas para evitar *loops* infinitos ou consumo excessivo de tokens. O agente deve interromper a execução quando uma ferramenta de saída final for invocada, quando o modelo retornar uma resposta final sem necessidade de novas chamadas, quando ocorrer um erro fatal ou quando o número máximo de turnos for atingido [OpenAI 2025]. Em sistemas como o LLM-NetCFG, limiares de tentativas são essenciais para encerrar o processo caso o modelo não produza uma configuração validada após múltiplas iterações [Lira et al. 2024].

3.5.3. Parâmetros de *Sampling* e Geração

Os parâmetros de *sampling* e geração são fundamentais para controlar a natureza estocástica dos LLMs. Eles regulam o equilíbrio entre criatividade e precisão técnica, sendo particularmente importantes em agentes de rede, nos quais se exige geração estruturada, previsível e auditável. Com base na literatura sobre as escalas e arquiteturas de agentes, os principais parâmetros são:

Temperature (T ou τ): controla o nível de incerteza e aleatoriedade durante a amostragem [Ha and Schmidhuber 2018]. Na função *softmax*, a temperatura ajusta a suavidade da distribuição de probabilidade [Sanh et al. 2020]. Valores baixos, como 0.2, tornam o modelo quase determinístico, favorecendo tarefas técnicas como geração de arquivos de configuração [Ha and Schmidhuber 2018, Lira et al. 2024]. Valores altos aumentam a diversidade, mas elevam o risco de alucinações e comportamento imprevisível [Ha and Schmidhuber 2018, Ouyang et al. 2022].

Top-k: limita a seleção aos k tokens mais prováveis em cada passo de geração. Essa restrição reduz a chance de o modelo escolher tokens irrelevantes da cauda da distribuição [Du et al. 2022]. Em cenários de chamada de ferramentas, por exemplo, pode-se exigir que um token de ação esteja entre os mais prováveis antes de permitir uma invocação [Schick et al. 2023].

Top-p (*nucleus sampling*): seleciona dinamicamente o menor conjunto de tokens cuja probabilidade acumulada atinge um valor p, como 0.9. Trata-se de uma alternativa mais flexível ao Top-k, pois adapta a janela de seleção conforme a confiança do modelo [Du et al. 2022].

Repeat penalty: penaliza tokens já emitidos, reduzindo a probabilidade de repetição excessiva. Embora nem sempre descrito formalmente na literatura arquitetural, é um parâmetro operacional útil para evitar *loops* verbais, comandos redundantes ou respostas

circulares.

Max tokens: define o limite máximo de tokens que o modelo pode gerar em uma única resposta. Em agentes de rede, esse parâmetro ajuda a controlar consumo de recursos e evitar saídas excessivamente longas quando o formato desejado é, por exemplo, apenas um objeto JSON [Lira et al. 2024, Kaplan et al. 2020].

Seed: a semente aleatória é importante para reprodutibilidade experimental. Quando suportada pelo sistema, permite reproduzir saídas sob as mesmas condições de *prompt* e parâmetros. Isso é relevante para auditoria e comparação de comportamento em cenários automatizados [Kaplan et al. 2020].

Uma configuração inicial recomendada para agentes de rede é:

```
{
  "temperature": 0.2,
  "top_p": 0.9,
  "top_k": 40,
  "repeat_penalty": 1.1
}
```

A configuração apresentada prioriza o determinismo, reduz ruído de geração e favorece comportamento previsível em ambientes críticos de infraestrutura [Ha and Schmidhuber 2018, Du et al. 2022].

3.5.4. Determinismo versus Criatividade

No projeto de agentes inteligentes, o parâmetro *temperature* atua como um regulador entre determinismo, isto é, escolha das respostas mais prováveis, e criatividade, entendida como exploração de alternativas menos prováveis. Embora a criatividade possa ser desejável em tarefas abertas, a automação de redes exige comportamento previsível, estável e repetível.

O uso de valores baixos de *temperature*, tipicamente entre 0.0 e 0.2, é recomendado nesse domínio por diversas razões técnicas.

Minimização de alucinações e erros de sintaxe: o aumento da aleatoriedade favorece a produção de comandos incorretos, estruturas inválidas ou interpretações técnicas imprecisas [Ouyang et al. 2022, Bandara et al. 2025]. Em redes, uma configuração errada de ACL, VLAN ou protocolo de roteamento pode gerar interrupções graves ou vulnerabilidades de segurança [Lira et al. 2024, Huang et al. 2023]. Temperaturas baixas mantêm o modelo mais próximo dos padrões sintáticos e operacionais mais prováveis [Ha and Schmidhuber 2018].

Previsibilidade e consistência operacional: com temperaturas elevadas, o mesmo estado de rede e o mesmo *prompt* podem produzir respostas diferentes em execuções sucessivas [Elkael et al. 2026]. Em ambientes críticos, espera-se comportamento repetível e auditável. Assim, configurar $\tau \approx 0$ aproxima o modelo de um sistema de decisão mais estável [Ha and Schmidhuber 2018].

Geração de formatos estruturados: agentes de rede frequentemente atuam como geradores de JSON, comandos CLI ou scripts. Nesses casos, estratégias próxi-

mas do *greedy decoding*, equivalentes a *temperature* baixa ou zero, tendem a produzir melhor aderência estrutural do que configurações mais criativas [Madaan et al. 2023, Bandara et al. 2025].

Segurança e estabilidade: aumentar a variância do processo decisório amplia a probabilidade de caminhos imprevisíveis e falhas difíceis de diagnosticar [Ha and Schmidhuber 2018]. Em arquiteturas ZSM e ambientes de automação crítica, a estabilidade do sistema depende de modelos que operem com baixa variabilidade e cujas saídas possam ser verificadas por ferramentas externas, como o Batfish, antes da aplicação [Elkael et al. 2026, Lira et al. 2024].

Em síntese, na IA Agentic aplicada a redes, a criatividade tende a representar risco operacional, enquanto o objetivo central é transformar o LLM em um executor de alta fidelidade, capaz de mapear intenções em comandos técnicos com o menor desvio possível [Elkael et al. 2026, Bandara et al. 2025].

3.5.5. Exemplos de *System Prompts* e *Prompts* de Usuário

Os princípios anteriores tornam-se mais claros quando traduzidos em instruções concretas. O modelo a seguir exemplifica um *system prompt* para um agente configurador de redes Linux:

```
IDENTIDADE:
Você é um agente de configuração de redes Linux. Sua única
função é traduzir instruções em linguagem natural para
sequências de comandos ip(8) válidos. Você não fornece
explicações, opiniões ou texto livre.

ESCOPO:
Você opera exclusivamente sobre: namespaces de rede,
interfaces veth, bridges Linux, VLANs (802.1Q), rotas
estáticas IPv4/IPv6 e túneis VXLAN. Recuse qualquer tarefa
fora desse escopo com a mensagem:
{"error": "fora do escopo operacional"}.

FORMATO DE SAÍDA OBRIGATÓRIO:
Retorne sempre um objeto JSON com a estrutura:
{
  "plan": ["passo 1", "passo 2", ...],
  "commands": ["cmd1", "cmd2", ...],
  "validation": ["cmd_verificação1", ...]
}
Nenhum texto fora desse JSON é permitido.

RESTRIÇÕES DE SEGURANÇA:
- Nunca emita comandos que removam interfaces em uso (estado UP).
- Nunca modifique a tabela de roteamento principal (tabela 254)
  sem flag explícito confirm=true no input.
- Limite o número de comandos por resposta a 20.
```

```
CRITÉRIO DE PARADA:
Se não for possível completar a tarefa com os dados fornecidos,
retorne: {"error": "informações insuficientes", "missing": [...]}.
```

Uma vez fixado esse *system prompt*, os *prompts* de usuário podem ser curtos, diretos e específicos. Alguns exemplos incluem:

Namespaces e veth:

```
"Crie ns1 e ns2 conectados por par veth.
Atribua 10.0.1.1/24 a ns1 e 10.0.1.2/24 a ns2."
```

Bridge Linux:

```
"Crie bridge br0 e adicione veth0 e veth1 como portas."
```

VLAN 802.1Q:

```
"Crie VLAN ID 10 sobre eth0 com endereço 192.168.10.1/24
e VLAN ID 20 com endereço 192.168.20.1/24."
```

Roteamento estático IPv4/IPv6:

```
"Em ns1, adicione rota para 10.0.2.0/24 via 10.0.1.2.
Adicione também rota IPv6 para fd00:2::/64 via fd00:1::2."
```

VXLAN:

```
"Crie túnel VXLAN VNI 100 entre 192.168.1.10 (local) e
192.168.1.20 (remoto). Endereço overlay: 10.100.0.1/24."
```

Esse padrão, baseado em *prompts* de usuário curtos e imperativos combinados com um *system prompt* restritivo, tende a produzir saídas JSON estruturadas que podem ser processadas diretamente por um executor Python, sem necessidade de *parsing* adicional. A separação entre identidade e restrições, definidas no *system prompt*, e a tarefa específica, definida no *prompt* de usuário, é o que torna possível reutilizar o mesmo agente em diferentes topologias sem reescrever suas políticas de segurança a cada invocação [OpenAI 2025, Lira et al. 2024].

3.6. Uso de RAG para Automação e Gerenciamento de Redes

Modelos de linguagem demonstram elevado desempenho em tarefas que envolvem a geração e a interpretação de longas cadeias de texto. Esse grau de sofisticação permite que modelos maiores e especializados construam linhas de raciocínio complexas e concluam tarefas com capacidades que, em certos contextos, se aproximam das humanas. No entanto, a natureza probabilística desses modelos pode produzir comportamentos indesejados, como erros sintáticos, semânticos e factuais. Esse cenário impõe um desafio significativo à implementação segura de sistemas agênticos baseados em LLMs ou SLMs em ambientes críticos.

Esta seção descreve a implementação e a importância da Geração Aumentada por Recuperação (*Retrieval-Augmented Generation* – RAG) como mecanismo fundamental para ampliar a precisão, a atualidade e a confiabilidade de agentes inteligentes aplicados ao gerenciamento de redes.

3.6.1. O Problema das Alucinações

Durante a inferência, modelos de linguagem operam ao estimar a probabilidade condicional de cada token em uma sequência para produzir a saída final. Embora essa mecânica favoreça fluidez textual, ela também pode conduzir a resultados factualmente incorretos, isto é, situações em que o modelo gera informações falsas, mas mantém um tom de elevada confiança. Esse fenômeno, amplamente conhecido como alucinação, manifesta-se quando o conteúdo gerado diverge da realidade ou da lógica interna do próprio texto. Tais falhas geralmente decorrem de dois cenários: desvios no raciocínio durante a geração ou tentativas do modelo de compensar a ausência de informações externas às quais ele não tem acesso [Zhou et al. 2024, Lira et al. 2024].

Em sistemas críticos, essa característica representa um dos maiores obstáculos à adoção de LLMs. No contexto de telecomunicações e redes de computadores, uma alucinação pode resultar em configurações inválidas, violações de políticas operacionais, erros em listas de controle de acesso (ACLs) e falhas de segurança com impacto sistêmico [Lira et al. 2024, Huang et al. 2023]. Técnicas de adaptação, como *Instruction Tuning* (IT), alinhamento com *Reinforcement Learning from Human Feedback* (RLHF) e *Continued Pretraining* (CPT), têm sido empregadas para reduzir a ocorrência de alucinações. Contudo, esses métodos costumam demandar alto custo computacional, com necessidade de GPUs de grande porte ou elevado consumo de créditos em provedores de nuvem.

Nesse contexto, a Geração Aumentada por Recuperação (RAG) surge como uma solução mais simples e economicamente viável para mitigar alucinações em aplicações de gerenciamento de redes. O RAG reduz esse risco ao ancorar o raciocínio do agente em evidências externas verificáveis, como RFCs, manuais de equipamentos, documentação operacional e logs de estado da rede [Chowa et al. 2026, de Lamo Castrillo et al. 2025]. Dessa forma, torna-se possível separar a base de conhecimento do mecanismo de geração, o que aumenta a flexibilidade para adicionar, remover ou atualizar conjuntos de documentos, como novas regulamentações, protocolos e manuais de dispositivos presentes na infraestrutura.

3.6.2. Geração Aumentada por Recuperação

A Geração Aumentada por Recuperação (RAG) é uma técnica que expande as capacidades de Modelos de Linguagem de Grande Escala (LLMs) ao permitir o acesso a fontes de dados externas e dinâmicas, eliminando a necessidade de reentreinamento contínuo [Lewis et al. 2020, Boateng et al. 2024a, Gao et al. 2023]. No ecossistema de redes de computadores onde protocolos, normas RFC e manuais técnicos evoluem aceleradamente, o RAG transfigura o agente de um gerador de texto genérico em um sistema especializado e fundamentado em evidências [Gao et al. 2023]. Essa arquitetura mitiga limitações intrínsecas aos modelos paramétricos, como a obsolescência do conhecimento e a ocorrência de alucinações [Jano 2024, Gao et al. 2023]. Ao acoplar modelos pré-treinados a módulos de recuperação não paramétricos, o sistema extrai informações contextuais durante a inferência, assegurando precisão técnica em consultas sobre infraestrutura e normas de rede [Lewis et al. 2020].

O alicerce funcional do RAG reside na Recuperação de Informação (IR), campo dedicado a prover o acesso eficiente a conteúdos de interesse mediante a organi-

zação e representação de itens, como registros de tráfego e documentação técnica [Manning et al. 2008]. Um sistema de recuperação opera para satisfazer uma necessidade informacional, formalizada por meio de uma consulta (*query*) [Manning et al. 2008]. O objetivo central consiste em maximizar a recuperação de documentos pertinentes, minimizando a inclusão de itens irrelevantes [Manning et al. 2008]. A eficácia dessa etapa é mensurada pelas métricas de precisão e revocação; em arquiteturas RAG, qualquer falha no componente de recuperação compromete a fidelidade da resposta final, podendo induzir o LLM a erros factuais críticos [Jano 2024, Gao et al. 2023].

No contexto da engenharia de redes, a recuperação pode ser abordada sob duas perspectivas complementares: lexical e semântica. A busca lexical fundamenta-se na correspondência exata de termos, sendo indispensável para localizar identificadores específicos, códigos de erro ou comandos de configuração [Anthropic 2024]. Atualmente, o algoritmo Okapi BM25 estabelece-se como o padrão para essa modalidade ao introduzir a normalização pelo comprimento dos documentos [Robertson and Zaragoza 2009]. A pontuação de relevância (score) entre um documento D e uma consulta Q é obtida por:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}, \tag{1}$$

onde $f(q_i,D)$ representa a frequência do termo q_i no documento D, $|D|$ é a extensão do documento e avgdl denota a média de comprimento da coleção [Robertson and Zaragoza 2009]. O componente de Frequência Inversa do Documento (IDF) é calculado conforme:

$$\text{IDF}(q_i) = \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right), \tag{2}$$

onde N é o total de documentos e $n(q_i)$ o número de documentos que contêm o termo específico [Robertson and Zaragoza 2009].

Enquanto o BM25 foca na sintaxe, a busca semântica fundamenta-se na representação do significado por meio de vetores densos, ou *embeddings* [Jano 2024]. Estes vetores são gerados por redes neurais que mapeiam o texto em um espaço latente de alta dimensionalidade [Lewis et al. 2020]. A similaridade entre os conceitos é quantificada via similaridade de cosseno:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}, \tag{3}$$

onde $\mathbf{A}$ e $\mathbf{B}$ representam os vetores de *embedding* em comparação [Jano 2024].

Para viabilizar essa operação em larga escala, utilizam-se bancos de dados vetoriais, que otimizam o armazenamento e a busca através de algoritmos de Busca de Vizinhos Mais Próximos Aproximados (ANN) [Pan et al. 2024]. A Tabela 3.7 sintetiza as principais soluções de mercado e suas características de desempenho.

Tabela 3.7. Principais Bancos de Dados Vetoriais

Banco de Dados	Natureza	Algoritmos	Escalabilidade	Latência (1M)
Milvus	Open-source	HNSW, DiskANN	Bilhões	50-80ms
Pinecone	Gerenciado	HNSW	Alta	40-50ms
Weaviate	Open-source	HNSW	Alta	50-70ms
FAISS	Biblioteca	HNSW, IVF	Média	10-20ms
Qdrant	Open-source	HNSW (Rust)	Alta	30-40ms

O penhasco dos 20000 documentos e o RAG agêntico

Evidências empíricas acumuladas entre 2024 e 2025 apontam para um padrão preocupante: grande parte das implementações corporativas de RAG falhou em seu primeiro ano de operação em produção, tornando-se uma das principais fontes de falha em sistemas de IA empresariais [Singh et al. 2025]. Uma das explicações propostas para esse fenômeno é o chamado *"penhasco dos 20000 documentos"*: sistemas RAG convencionais, baseados em recuperação simples dos trechos mais relevantes para alimentar um LLM, tendem a perder controle epistêmico, coerência semântica e fidelidade informacional quando o volume documental ultrapassa determinado limiar, mesmo que funcionem adequadamente em demonstrações com bases reduzidas.

Como resposta, a indústria passou a adotar o chamado **RAG agêntico**. Em vez de realizar recuperação passiva, o agente passa a orquestrar dinamicamente ferramentas de busca, manter trilhas de memória de longo prazo, formular planos iterativos de recuperação e refletir recursivamente sobre ações anteriores [Singh et al. 2025]. No domínio de redes, isso significa que o agente não apenas recupera a RFC relevante para uma configuração BGP; ele também verifica ativamente se o documento recuperado é consistente com o estado atual da topologia, busca documentação adicional quando a confiança é baixa e registra o raciocínio de recuperação para fins de auditoria.

O **RCR-Router** [Liu et al. 2025] exemplifica otimizações específicas para RAG em sistemas multiagente. Ao reutilizar resultados de recuperação entre agentes que compartilham documentos-base, mas possuem prefixos de consulta distintos, o sistema reduz em mais de 30% o número de tokens consumidos sem perda mensurável de qualidade. Validado em *benchmarks* como HotPotQA, MuSiQue e 2WikiMultihop, que exigem raciocínio em múltiplos saltos, o RCR-Router sugere que a eficiência do processo de recuperação é um eixo de otimização tão relevante quanto a qualidade do modelo gerador.

3.6.3. Entendendo *Embeddings*: Texto, Imagem e Multimodalidade

A base tecnológica do RAG reside nos *embeddings*, isto é, representações numéricas vetoriais de informações em espaços de alta dimensionalidade [de Lamo Castrillo et al. 2025].

Embeddings de texto: convertem palavras, sentenças ou documentos em vetores que capturam relações semânticas, permitindo que o sistema recupere conteúdo por significado, e não apenas por correspondência exata de palavras-chave [Karpukhin et al. 2020, Boateng et al. 2024b].

Embeddings de imagem: utilizam codificadores visuais, como Redes Neurais

Convolucionais (CNNs) ou *Vision Transformers* (ViTs), para extrair características representativas de diagramas, topologias e gráficos de rede [de Lamo Castrillo et al. 2025].

Embeddings multimodais: modelos multimodais avançados aprendem um espaço vetorial unificado, no qual dados textuais e visuais são convertidos em representações compatíveis. Isso permite ao sistema relacionar, por exemplo, uma descrição textual de falha a um diagrama de arquitetura ou a uma imagem operacional [de Lamo Castrillo et al. 2025, Bommasani et al. 2022].

3.6.4. *Pipeline* Completo de RAG

Um sistema RAG operacional para redes geralmente segue um fluxo de trabalho estruturado, composto pelas seguintes etapas:

***Embedding* (indexação)**: documentos técnicos, RFCs, manuais e registros operacionais são segmentados em pequenos trechos (*chunks*) e convertidos em vetores por meio de um codificador (*encoder*) [Borgeaud et al. 2022, Karpukhin et al. 2020].

Recuperação (*retrieval*): quando o operador formula uma consulta, o sistema converte essa consulta em vetor e realiza uma busca por similaridade, frequentemente por Máximo Produto Interno (MIPS), em um banco de dados vetorial, como o FAISS, a fim de localizar os trechos mais relevantes [Lewis et al. 2021, Karpukhin et al. 2020].

***Reranker* (re-ranqueamento)**: componente opcional, mas frequentemente crucial, que utiliza modelos mais refinados, como arquiteturas com atenção cruzada, para avaliar a relevância real entre a pergunta e os documentos recuperados, reorganizando-os para priorizar as passagens mais úteis [Karpukhin et al. 2020, de Lamo Castrillo et al. 2025].

Contextualização: os documentos recuperados são concatenados à consulta original do usuário, formando um *prompt* enriquecido com contexto documental [Lewis et al. 2021, Kong et al. 2025].

Geração: por fim, o modelo de linguagem combina sua memória paramétrica, derivada do treinamento, com a memória não paramétrica, proveniente dos documentos recuperados, para gerar uma resposta mais precisa, atualizada e fundamentada [Lewis et al. 2021, Chowa et al. 2026].

3.6.5. Comparação de *Cmbeddings*: SLM versus LLM

A escolha do modelo de *embedding* afeta diretamente a eficiência, o custo e a privacidade do sistema aplicado à rede.

***Embeddings* baseados em SLMs**: modelos compactos, como BERT-base, frequentemente são empregados como codificadores em sistemas RAG por serem leves, rápidos e adequados à execução local ou em servidores de borda [Lewis et al. 2021, Borgeaud et al. 2022]. Esses modelos viabilizam indexação e recuperação de grandes volumes de informação com custo operacional relativamente baixo.

***Embeddings* baseados em LLMs**: modelos maiores tendem a oferecer compreensão semântica mais sofisticada para relações complexas e dependências de longo alcance. Em contrapartida, exigem recursos computacionais substanciais, como GPUs de alto de-

sempenho, e podem introduzir latência incompatível com tarefas de rede em tempo real [Boateng et al. 2024b, Zhou et al. 2024].

Em síntese, o uso de RAG com modelos locais, como SLMs ou LLMs compactos, permite que operadores de rede preservem a privacidade dos dados e mantenham a base de conhecimento constantemente atualizada por meio da simples substituição ou ampliação dos documentos de referência, sem necessidade de reentreinar continuamente o modelo principal [Lira et al. 2024, Lewis et al. 2021].

3.7. Arquitetura Prática de Implementação: Python + SLM/LLM (Ollama/ OpenWebUI)

Esta seção apresenta a arquitetura técnica para a implementação de um sistema de IA aplicado a redes, utilizando Python como orquestrador principal e integrando SLMs e LLMs. A arquitetura proposta adota uma abordagem modular, na qual os componentes de decisão, recuperação de conhecimento, validação e execução são explicitamente separados, favorecendo segurança, auditabilidade e flexibilidade.

Essa organização permite a integração com soluções mais amplas, uma vez que cada módulo pode ser evoluído ou substituído de forma independente. Além disso, a separação de responsabilidades facilita a validação individual dos componentes, característica especialmente relevante em arquiteturas baseadas em cadeias de decisão.

Nesse contexto, diferentes modelos podem ser utilizados de forma complementar, explorando suas especialidades. Por exemplo, um modelo pode atuar na interpretação da intenção do usuário, outro na análise aprofundada e um terceiro na geração de ações ou respostas finais, formando um fluxo progressivo e encadeado de processamento.

3.7.1. Diagrama Arquitetural

A arquitetura baseia-se em uma estrutura modular, na qual o Python atua como camada de integração entre os subsistemas. O diagrama arquitetural compreende os seguintes blocos:

Módulo orquestrador/executor/auditor (Python): centraliza a lógica de negócio, gerencia o fluxo de mensagens e realiza a integração com os demais componentes. Também é responsável por interpretar as saídas do modelo, executar comandos de forma controlada, coletar resultados e realimentar o modelo. Adicionalmente, implementa mecanismos de registro (logging) para auditoria e rastreabilidade.

Módulo de agente (SLM/LLM): constitui o núcleo cognitivo do sistema, responsável por interpretar intenções, raciocinar sobre o estado do ambiente e gerar ações ou comandos técnicos [Lira et al. 2024, Elkael et al. 2026]. A arquitetura permite avaliar diferentes modelos, bem como ajustar *prompt* de sistema, parâmetros de geração e integração com RAG.

Repositório de configuração e base de conhecimento: armazena configurações aprovadas e documentação técnica. Esse repositório é utilizado por mecanismos de RAG, geralmente suportados por bancos vetoriais, permitindo recuperação contextual baseada em *embeddings*.

3.7.2. Mecanismos Operacionais

A comunicação entre os módulos ocorre, preferencialmente, por meio de APIs REST. O uso de objetos JSON estruturados permite que o orquestrador em Python realize requisições e processe respostas de forma consistente.

Embora a literatura recomende a utilização de JSON como formato padrão de saída para os modelos enviarem para execução, neste minicurso adota-se, para fins didáticos, um formato alternativo mais simples, que facilita o *parsing* automático das respostas e reduz ambiguidades na integração com o ambiente de execução.

Para mitigar riscos operacionais e de segurança, a execução de ações deve seguir o princípio do privilégio mínimo:

Sandboxing: o executor deve operar em ambientes isolados, como containers Docker ou máquinas virtuais efêmeras, com acesso restrito a recursos do sistema [Du et al. 2026].

Controle de acesso via API: recomenda-se o uso de tokens com permissões limitadas, garantindo que o agente acesse apenas recursos previamente autorizados.

Funções determinísticas: operações críticas devem ser delegadas a funções implementadas em Python, evitando dependência direta do comportamento estocástico do modelo [Bandara et al. 2025].

A rastreabilidade depende de uma camada de observabilidade que registre decisões, ações e evidências:

Logs de execução: registram o fluxo de decisões, chamadas de ferramentas e resultados observados [Du et al. 2026, Kong et al. 2025].

Auditoria imutável: em cenários críticos, recomenda-se o uso de mecanismos de armazenamento imutável para suportar análises forenses [Kong et al. 2025].

O sistema deve implementar mecanismos de controle para lidar com falhas:

Limiar de iterações: define um número máximo de tentativas para evitar *loops* infinitos [Lira et al. 2024].

Escalonamento humano: em casos de falha persistente ou operações críticas, o controle deve ser transferido para um operador humano [OpenAI 2025].

3.7.3. Requisitos de Infraestrutura

Os requisitos variam conforme o modelo adotado e o cenário de uso:

Processamento: SLMs podem operar em hardware moderado, enquanto LLMs e *pipelines* RAG mais complexos demandam GPUs [Lira et al. 2024, Elkael et al. 2026].

Armazenamento: bases vetoriais, modelos e artefatos intermediários podem exigir alguns terabytes de armazenamento [Lira et al. 2024].

Orquestração: o uso de Kubernetes facilita escalabilidade, resiliência e alta disponibilidade [Bandara et al. 2025].

Esta arquitetura define um agente como um *pipeline* composto por componentes

de decisão, memória, recuperação de conhecimento e execução. O modelo de linguagem atua como mecanismo de decisão, enquanto o executor em Python realiza ações e retorna evidências observáveis, formando um ciclo iterativo de decisão e validação.

3.7.4. Visão Geral do *Pipeline* de Agentes

Em arquiteturas modernas, um único modelo raramente atende a todos os requisitos. Assim, diferentes modelos podem assumir papéis específicos. O **modelo de decisão** interpreta tarefas e define ações; o **modelo de transformação** realiza sumarização, normalização ou extração estruturada; e o **modelo de embeddings** suporta a busca semântica em sistemas RAG.

O comportamento do sistema é influenciado por diferentes fatores, incluindo o **prompt de sistema**, os **parâmetros de geração**, as **estratégias de redução de alucinação** e os mecanismos de **adaptação do modelo**.

3.7.5. Componentes da *Stack* do Minicurso

A *stack* é composta por três elementos principais:

1. **Ollama**[2]: runtime local para execução de modelos de linguagem, com API de inferência.
2. **OpenWebUI**[3]: camada de configuração, experimentação e integração com RAG e facilitadade de ajustes.
3. **Executor em Python**: responsável pela execução controlada de ações e fechamento do ciclo de feedback.

A Figura 3.2 ilustra a integração entre esses componentes.

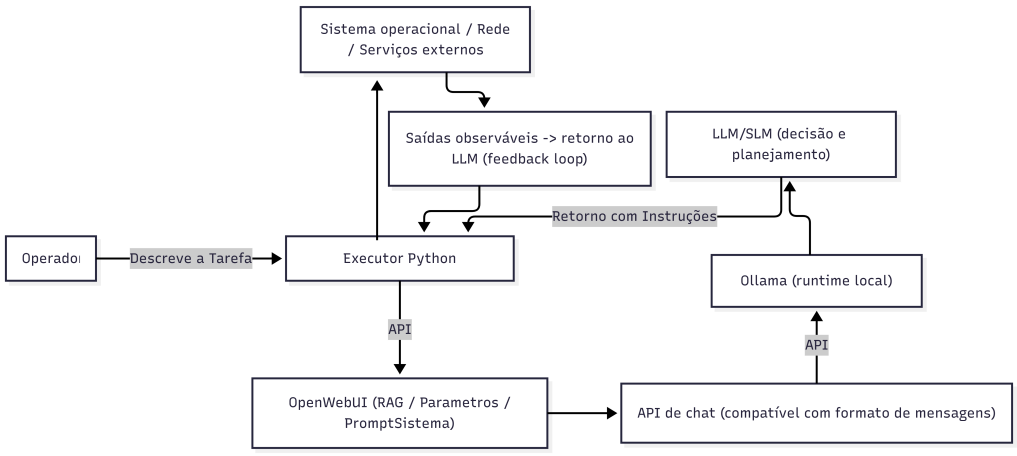

Figura 3.2. Arquitetura geral do agente utilizado no minicurso, destacando a separação entre decisão (LLM) e execução (Python)

[2]https://ollama.com
[3]https://openwebui.com

3.7.6. *Feedback Loop* e Separação entre Decisão e Execução

O modelo não executa ações diretamente. Em vez disso, propõe ações com base no estado observado. O executor realiza a ação, coleta evidências e devolve os resultados ao modelo, que decide o próximo passo.

Esse ciclo iterativo caracteriza um *feedback loop* controlado, fundamental para confiabilidade em ambientes de infraestrutura.

3.7.7. Executor em Python: Responsabilidades

O executor deve garantir previsibilidade e segurança por meio de: validação de comandos; controle de tempo de execução; captura de saídas; retorno estruturado; limitação de iterações; critérios de parada.

3.7.8. Síntese

A arquitetura proposta é adequada para o minicurso por três razões principais: modularidade; reprodutibilidade; segurança e controle.

Ao final, o participante será capaz de construir um agente funcional que executa ações controladas, interpreta resultados e produz diagnósticos baseados em evidências reais e tem a facilidade de ajustar, experimentar o seu agente de forma intuitiva.

3.8. Implementação Completa de um Agente Configurador de Redes

O agente foi implementado em Python 3.10+, utilizando o Ollama como ambiente de execução de modelos SLM/LLM e o OpenWebUI como interface de interação. A execução dos comandos requer privilégios administrativos no sistema operacional.

A integração entre OpenWebUI e Ollama é realizada via *Docker Compose*, permitindo que os modelos sejam executados localmente e acessados diretamente pela interface Web, sem necessidade de configuração adicional de APIs externas.

A estrutura do projeto segue uma separação clara entre orquestração, base de conhecimento e execução e tem as seguintes necessidades[4]:

```
agnet.py            # orquestrador do ciclo deliberativo
docker-compose.yaml
ollama-*.sh         # scripts de gerenciamento de modelos
rag-estruturado/    # base de conhecimento (RAG)
   01-criacao-redes.md
   02-bridge.md
   03-enderecamento-redes.md
   04-criacao-host.md
   05-veth.md
   06-enderecamento-hosts.md
   07-template.md
   08-testar.md
```

A base RAG desempenha papel central na redução de alucinações, fornecendo

[4]https://gitlab.ic.unicamp.br/minicurso-sbrc2026/agnet

exemplos concretos e padrões válidos de comandos `ip`. O agente consulta esses documentos para fundamentar suas decisões antes da geração de comandos.

3.8.1. Configuração do Modelo no OpenWebUI

No OpenWebUI, o agente é criado a partir da clonagem de um modelo previamente carregado no Ollama (por exemplo, `ministral:3b` ou `qwen3.5:9b`). A clonagem permite especializar o comportamento do modelo por meio de um *prompt de sistema*, sem alterar o modelo original.

Após a clonagem do modelo no OpenWebUI, define-se um *prompt de sistema* que estabelece o comportamento do agente. Esse *prompt* atua como um mecanismo de restrição operacional, impondo limites claros de atuação, formato de saída e estratégia de decisão.

Diferentemente de interações convencionais com modelos de linguagem, o *prompt* foi projetado para transformar o modelo em um agente determinístico, orientado exclusivamente à execução de comandos e baseado na observação do estado do sistema.

O *promp* utilizado é apresentado a seguir:

```
Você é um agente de configuração de redes Linux virtuais.

Escopo operacional permitido:
- network namespaces
- interfaces veth
- bridges Linux
- endereçamento IP
- rotas IPv4
- testes de conectividade com ping

Ambiente de execução:
- Um executor em Python executa exatamente os comandos que
  você retornar
- Você nunca executa comandos diretamente
- Após cada execução, você recebe a saída real dos comandos
- Toda decisão deve ser baseada apenas nessa saída real

Objetivo:
- Construir, reconciliar, validar e corrigir topologias Linux
  virtuais de forma incremental e segura

Princípios obrigatórios:
- Nunca assumir o estado atual do sistema
- Operar incrementalmente
- Executar apenas o próximo passo mínimo
- Nunca duplicar recursos

Formato de saída obrigatório:
- Apenas comandos dentro de (( ))
- Um comando por bloco
```

```
Formato:
((comando))

Sucesso:
((config finalizado com sucesso))

Falha:
((config sem solução))

Consultas iniciais obrigatórias:
((ip netns list))
((ip link show))
((ip addr show))
((ip link show type bridge))

Validação antes de sucesso:
- namespaces existem
- interfaces e bridges UP
- lo UP
- IP correto
- conectividade com ping -c 3

Restrições:
- nunca modificar interfaces físicas
- nunca executar comandos destrutivos
- nunca usar conhecimento externo
```

O *prompt de sistema* foi projetado para obrigar o modelo a responder exclusivamente com comandos delimitados por `((...))`, formato que é posteriormente processado pelo executor Python por meio de expressões regulares.

Esse tipo de engenharia de *prompt* reduz significativamente respostas não determinísticas e força o modelo a atuar como um executor simbólico, alinhado ao paradigma de agentes deliberativos. Além disso, o uso de um formato estruturado de saída ((...)) permite integração direta com o executor Python, eliminando a necessidade de parsing ambíguo.

3.8.2. Ajuste de Parâmetros de Geração

Para garantir comportamento determinístico, os parâmetros de geração são configurados diretamente no OpenWebUI:

- **Temperatura**: 0.0–0.2
- **Top-p**: próximo de 1.0
- **Max tokens**: limitado para evitar respostas longas
- **Repetition penalty**: leve ajuste para evitar redundância

Essa configuração reduz a variabilidade das respostas e torna o agente mais previ-

sível.

3.8.3. Configuração do RAG e *Embeddings*

O OpenWebUI permite a ingestão de documentos para suporte a *Retrieval-Augmented Generation* (RAG). Os arquivos do diretório `rag-estruturado/` são carregados na interface e indexados automaticamente e durante a execução, o modelo consulta essa base para recuperar padrões válidos de comandos, reduzindo alucinações e melhorando a precisão das ações geradas. A escolha do modelo de *embeddings* impacta diretamente a qualidade da recuperação semântica. Recomenda-se a utilização de modelos leves executados localmente via Ollama, garantindo baixa latência e independência de serviços externos, porém o próprio OpenWebUI se inicia com um modelo integrado que pode ser alterado.

3.8.4. Fluxo Integrado com RAG

Após a configuração, o fluxo de execução do agente é:

```
Entrada do usuário
        ↓
Consulta RAG (embeddings)
        ↓
Interpretação pelo modelo
        ↓
Geração de comandos (( ))
        ↓
Execução no sistema
        ↓
Retorno da saída real
        ↓
Novo ciclo
```

Essa integração entre modelo de linguagem, base de conhecimento e execução real caracteriza um sistema de agentes completo, capaz de operar de forma iterativa e orientada ao estado do sistema.

3.8.5. Integração do OpenWebUI com o Executor Python

Após a configuração do modelo clonado no OpenWebUI, sua integração com o ambiente de execução é realizada por meio de um orquestrador em Python, implementado no arquivo `agnet.py`. Esse componente atua como ponte entre o modelo de linguagem e o sistema operacional, sendo responsável por enviar instruções ao modelo, extrair os comandos gerados, executá-los localmente e retornar a saída real ao agente em um ciclo iterativo.

O `agnet.py` consome diretamente a API compatível com *chat completions* disponibilizada pelo OpenWebUI, utilizando uma chamada HTTP ao endpoint `/api/chat/completions`. Nesse processo, o script envia o histórico de mensagens contendo a instrução inicial do usuário e, nas iterações seguintes, as saídas observadas dos comandos previamente executados. Esse mecanismo permite ao modelo tomar decisões

fundamentadas exclusivamente no estado real retornado pelo sistema.

A integração é baseada nos seguintes parâmetros principais do executor:

- **API_URL**: endereço do endpoint do OpenWebUI;
- **API_KEY**: chave de autenticação para acesso à API;
- **MODEL**: nome do modelo clonado configurado no OpenWebUI;
- **NUMINTERACOES**: limite máximo de iterações do ciclo deliberativo;
- **LOG_FILE**: arquivo de log contendo histórico de mensagens e respostas.
- **INSTRUADIC**: instrução adicional ao LLM a cada interação, pode ser definir o numero da interação ou incluir informações relevantes.

No protótipo desenvolvido, o modelo configurado no OpenWebUI é referenciado diretamente pelo executor por meio da variável `MODEL`, permitindo que o mesmo agente especializado seja utilizado de forma programática. Assim, uma vez clonado e configurado na interface do OpenWebUI, o modelo passa a ser acessível pelo script Python sem necessidade de adaptações adicionais no mecanismo de inferência.

Envio da instrução inicial: O processo é iniciado com uma instrução em linguagem natural fornecida como argumento ao script. Essa instrução é adicionada ao histórico de mensagens e enviada ao modelo por meio da função `call_llm()`, que encapsula a requisição HTTP ao OpenWebUI.

Extração de comandos: A resposta do modelo é processada pela função `extract_commands()`, que identifica todos os comandos delimitados no formato `((...))`. Esse padrão é consistente com o *prompt de sistema* definido no OpenWebUI e garante separação clara entre raciocínio implícito do modelo e ações executáveis pelo sistema.

Execução local: Cada comando extraído é executado pelo executor com `subprocess.run()`, utilizando `shell=True`, captura de `stdout` e `stderr`, e retorno textual consolidado. O executor não interpreta semanticamente o comando: ele apenas executa exatamente a instrução retornada pelo modelo, preservando a separação entre decisão e ação.

Realimentação do modelo: Após a execução, a saída observada de cada comando é anexada ao histórico como nova mensagem do usuário. Esse retorno inclui tanto o comando executado quanto sua saída correspondente, seguido de uma instrução adicional solicitando ao agente os próximos passos ou a finalização da configuração. Dessa forma, o modelo recebe evidências concretas do estado da rede e pode decidir entre continuar, validar ou encerrar a interação.

Critérios de parada: O ciclo é encerrado quando o modelo retorna `((config finalizado com sucesso))`, indicando que a intenção foi satisfeita, ou `((config sem solução))`, indicando impossibilidade de prosseguir de forma segura. O processo também é interrompido quando o número máximo de interações é atingido, evitando laços infinitos.

3.8.6. Fluxo Operacional do Executor

O comportamento integrado entre OpenWebUI e executor Python pode ser resumido pelo fluxo da Figura textual abaixo:

```
[Usuário]
   ↓
prompt inicial
   ↓
[agnet.py]
   ↓
requisição HTTP para /api/chat/completions
   ↓
[OpenWebUI + modelo clonado]
   ↓
resposta com comandos ((...))
   ↓
[agnet.py]
   ↓
extração dos comandos
   ↓
execução via subprocess.run()
   ↓
captura de stdout/stderr
   ↓
envio da saída real ao modelo
   ↓
novo ciclo deliberativo
```

Esse fluxo caracteriza uma arquitetura de agente *in-the-loop*, em que o modelo não executa ações diretamente, mas delibera sobre observações reais retornadas por um executor externo. Essa separação é importante para garantir auditabilidade, controle e possibilidade de inserção de políticas adicionais de segurança no executor.

3.8.7. Exemplo de Configuração do Executor

A integração prática requer apenas que o nome do modelo configurado no OpenWebUI seja referenciado corretamente no executor. No protótipo implementado, isso é feito por meio da definição:

```
API_URL = "http://localhost:3000/api/chat/completions"
MODEL = "agnet-ministral"
```

Com isso, o comando:

```
python3 agnet.py "criar dois hosts em uma rede isolada e testar"
```

faz com que o orquestrador envie a solicitação ao modelo especializado, execute os comandos retornados e mantenha o ciclo de observação e correção até a conclusão da tarefa ou até atingir o limite de interações definido.

3.8.8. Interação com a API e gerenciamento de contexto

O agente utiliza o *endpoint* de *chat completions* do OpenWebUI, compatível com a API da OpenAI, permitindo integração com modelos locais ou remotos por meio de requisições HTTP em formato JSON. As requisições são realizadas via `POST` para `http://fqdn/api/chat/completions`, com autenticação via *Bearer Token* no cabeçalho (`Authorization`) e `Content-Type: application/json`.

O corpo da requisição contém o modelo e o histórico da interação no campo `messages`, estruturado como uma lista de mensagens com papéis (`role`) e conteúdos (`content`), representando o diálogo entre usuário e assistente. Um exemplo simplificado é apresentado a seguir:

```
{
  "model": "agnet-ministral",
  "messages": [
    {"role": "user", "content": "criar rede"},
    {"role": "assistant", "content": "((ip netns add h1))"},
    {"role": "user", "content": "Saida do comando...\n
    Quais os proximos passos?"}
  ]
}
```

Como o modelo não mantém estado entre chamadas, todo o contexto é explicitamente reenviado a cada requisição. O agente opera em um ciclo iterativo no qual cada resposta gerada é tratada como um comando, executada no sistema e, em seguida, seu resultado é reinserido no histórico. Dessa forma, o processo estabelece um fluxo contínuo de geração, execução e realimentação, possibilitando raciocínio incremental baseado no estado atual do ambiente e garantindo consistência ao longo da interação.

3.8.9. Aspectos de Segurança e Controle do Executor

Embora o modelo seja responsável por propor comandos, o controle efetivo da execução permanece no orquestrador Python. Essa arquitetura permite registrar todas as interações em arquivo de log, limitar o número máximo de ciclos, padronizar determinados comportamentos e, potencialmente, inserir filtros adicionais antes da execução dos comandos.

No protótipo atual, o executor já aplica uma normalização simples para comandos de `ping`, forçando o uso de `ping -c 3`. Esse mecanismo ilustra como políticas operacionais podem ser implementadas no executor para reforçar previsibilidade, repetibilidade e segurança.

Essa abordagem segue o princípio de *guardrails*, no qual o executor atua como uma camada de contenção, limitando o espaço de ações do agente e garantindo previsibilidade operacional.

Validação:

```
ip netns exec ns1 ping -c 2 10.0.0.2
```

A interpretação do resultado do `ping` permite ao agente confirmar a conectividade ou iniciar um processo de correção.

Controle do executor e normalização de comandos No executor implementado, comandos potencialmente contínuos, como o `ping`, são automaticamente normalizados para evitar execuções indefinidas. Por exemplo, qualquer chamada ao comando `ping` é transformada para incluir o parâmetro `-c 3`, limitando o número de pacotes enviados:

```
ping 10.0.0.2 → ping -c 3 10.0.0.2
```

Essa normalização é implementada diretamente no executor Python, conforme ilustrado a seguir:

```
cmd = re.sub(r'ping\s+(\d{1,3}(?:\.\d{1,3}){3})',
             r'ping -c 3 \1', cmd)
```

Esse mecanismo evita que o agente gere comandos que possam bloquear a execução (por exemplo, *ping* contínuo) e ilustra como políticas de segurança e controle podem ser incorporadas no executor.

De forma análoga, o executor pode ser estendido para filtrar ou bloquear comandos potencialmente destrutivos, reforçando a separação entre o processo de decisão do modelo e a execução controlada no sistema operacional.

Em conjunto, OpenWebUI, RAG, modelo clonado e executor Python compõem uma arquitetura de agente prático, na qual o modelo interpreta intenções e propõe ações, enquanto o executor observa, aplica e devolve o estado real do sistema para novo ciclo de decisão.

3.9. Segurança, Riscos e Boas Práticas no Uso de Agentes em Redes

Esta seção analisa os riscos críticos associados à implantação de agentes inteligentes em infraestruturas de rede e apresenta estratégias fundamentais para projetar sistemas seguros, auditáveis e resilientes.

3.9.1. Análise de Riscos

A transição de modelos de linguagem estáticos para agentes autônomos amplia significativamente a superfície de ataque, pois esses sistemas não apenas geram texto, mas também possuem a capacidade de invocar ferramentas e interagir diretamente com o ambiente digital e, em alguns casos, com sistemas físicos [Kong et al. 2025].

Execução arbitrária e uso indevido de ferramentas. Diferentemente de LLMs tradicionais, agentes podem produzir impactos concretos no ambiente operacional, como operar serviços de forma incorreta, corromper bases de dados ou comprometer sistemas críticos de rede [Kong et al. 2025]. A execução de código gerado automaticamente, como scripts em Python, representa um risco severo, pois pode envolver acesso não autorizado ao sistema de arquivos, comandos de *shell* maliciosos ou importações inseguras [Derouiche et al. 2025].

***Prompt injection*.** Esse é um dos riscos mais relevantes e pode ser dividido em duas categorias principais [Kong et al. 2025]:

- **Direta**: ocorre quando o usuário fornece instruções destinadas a sobrescrever o *system prompt* original, por exemplo, com comandos do tipo "ignore todas as instruções anteriores", buscando subverter o comportamento do agente [Kong et al. 2025].
- **Indireta**: ocorre quando o agente consome dados de fontes externas, como documentos recuperados via RAG ou páginas web, que contêm comandos maliciosos disfarçados. Nesse caso, o agente pode ser induzido a executar ações não planejadas, como vazar variáveis de ambiente ou segredos de configuração [Kong et al. 2025, Du et al. 2026].

Escalada de privilégio (*authority abuse*). Em sistemas multiagente, um invasor pode induzir um agente de menor privilégio a repassar um contexto manipulado para um agente com permissões superiores [Kong et al. 2025]. Caso o agente receptor herde essa confiança implicitamente, sem verificar o escopo original de autorização, o atacante poderá acionar operações administrativas fora de sua permissão [Kong et al. 2025].

Erros operacionais críticos. As alucinações técnicas podem levar à geração de configurações sintaticamente corretas, porém logicamente desastrosas, como uma ACL que isole o roteador de gerenciamento [Lira et al. 2024, Huang et al. 2023]. Além disso, ataques de exaustão cognitiva podem induzir o agente a entrar em ciclos excessivos de raciocínio e tentativa, consumindo recursos computacionais e degradando a disponibilidade do serviço de gerenciamento [Kong et al. 2025].

3.9.2. Estratégias de Mitigação e Boas Práticas

Para mitigar esses riscos, a arquitetura deve ser construída sob uma filosofia de defesa em profundidade, combinando controles sistêmicos, restrições operacionais e mecanismos especializados de segurança.

Princípios de *Zero Trust*. A segurança não deve depender de perímetros fixos, mas de verificação contínua. No contexto de agentes, isso implica autenticação mútua entre agentes e ferramentas, isolamento de contexto e verificação de privilégios em cada etapa do fluxo de trabalho [Kong et al. 2025, Boateng et al. 2024b]. Cada ferramenta deve ser executada com o princípio do menor privilégio, em ambientes isolados, como contêineres Docker com restrições rígidas de rede e sistema de arquivos [Derouiche et al. 2025, Bandara et al. 2025].

***Classifier models* e *guardrails*.** O uso de classificadores de segurança dedicados permite detectar tentativas de *jailbreak* e injeção de *prompt* antes que atinjam o núcleo cognitivo do sistema [OpenAI 2025]. Esses *guardrails* funcionam como uma camada defensiva que valida tanto a entrada do usuário quanto a saída do modelo, garantindo que as respostas permaneçam alinhadas às políticas de segurança e privacidade, incluindo filtros de dados sensíveis [OpenAI 2025].

***Alignment* e modelos de segurança.** Estratégias como a IA Constitucional buscam tornar os modelos mais seguros por meio de princípios explícitos de comportamento e regras operacionais, permitindo que o agente recuse solicitações maliciosas e justifique tecnica-

mente sua recusa [Bai et al. 2022]. Além disso, a adoção de um consórcio de modelos pode aumentar a robustez do sistema: múltiplos LLMs produzem saídas independentes, posteriormente validadas por um agente verificador, reduzindo a probabilidade de que uma única alucinação ou viés seja propagado [Bandara et al. 2025, Du et al. 2026].

Auditoria e limites operacionais. A governança do sistema requer mecanismos claros de observabilidade e contenção:

- **Logs de auditoria**: é essencial manter registros imutáveis de entradas, saídas, chamadas de ferramentas e rastros de execução observáveis, a fim de viabilizar análises forenses e detecção de anomalias [Park et al. 2023, Kong et al. 2025].
- **Limites de execução**: devem ser estabelecidos limiares rígidos para falhas, tempo de execução e número de turnos. Se um agente exceder o número permitido de tentativas de configuração ou tentar realizar uma ação de alto risco, como desligar uma interface principal, o sistema deve pausar a operação e exigir intervenção humana (*Human-in-the-Loop*) [OpenAI 2025, Lira et al. 2024].
- **Separação de lógica**: funções puramente operacionais, como escrita em disco, *commits* ou alterações persistentes de estado, devem ser delegadas a funções determinísticas implementadas em código, por exemplo em Python, reservando o LLM apenas para etapas que exijam interpretação semântica e raciocínio linguístico [Bandara et al. 2025].

Ao integrar essas práticas, operadores de rede podem construir um ecossistema de NetOps 2.0 em que a automação inteligente seja equilibrada por mecanismos rigorosos de governança, segurança e controle [Lira et al. 2024, Bandara et al. 2025].

3.10. Conclusão e Perspectivas Futuras

Esta seção final consolida os principais aprendizados deste minicurso e discute perspectivas futuras para o gerenciamento de infraestruturas de rede, em um cenário no qual a convergência entre modelos de linguagem, agentes inteligentes e protocolos de rede redefine a autonomia operacional.

A trajetória do NetOps 2.0 representa a transição de sistemas baseados em scripts estáticos para ecossistemas de inteligência de rede unificada [Huang et al. 2023]. Nesse cenário, a IA generativa deixa de atuar apenas como assistente textual e passa a ocupar o papel de núcleo cognitivo de sistemas capazes de perceber, planejar e agir sobre a infraestrutura de forma autônoma ou semiautônoma [Kong et al. 2025, Elkael et al. 2026].

3.10.1. Consolidação da Arquitetura Multimodelo

O paradigma contemporâneo abandonou a dependência de modelos únicos e monolíticos em favor de fluxos de trabalho agênticos (*Agentic AI Workflows*) [Bandara et al. 2025]. Nessa arquitetura modular, responsabilidades são distribuídas entre componentes especializados, cada qual com papel definido no ciclo de operação:

- **Orquestradores e roteadores**: responsáveis por gerenciar o fluxo de tarefas e selecionar os recursos mais adequados a cada demanda [de Lamo Castrillo et al. 2025, Bandara et al. 2025].

- **Modelos de raciocínio e verificadores**: encarregados de sustentar consistência, segurança e confiabilidade por meio de planejamento, autorreflexão e validação sintática ou semântica [de Lamo Castrillo et al. 2025, Lira et al. 2024].
- **Modelos locais (SLMs)**: fundamentais para assegurar privacidade de dados e baixa latência em operações críticas executadas na borda (*edge*) ou em ambientes com maior restrição operacional [Lira et al. 2024].

3.10.2. O Papel Transformador da IA Generativa em Redes

LLMs e VLMs (*Vision-Language Models*) ampliaram significativamente a capacidade de processar dados não estruturados e multimodais, como telemetria, diagramas de topologia e logs complexos, permitindo uma compreensão mais ampla do estado da rede [de Lamo Castrillo et al. 2025, Zhou et al. 2024]. A principal inovação reside na interpretação de intenção (*intent-driven*), que reduz a distância entre objetivos de negócio e execução técnica, viabilizando que a rede se autoconfigure a partir de comandos expressos em linguagem natural [Huang et al. 2023, Lira et al. 2024].

3.10.3. Tendências e Desdobramentos para o Horizonte 2026–2030

A evolução da automação de redes no horizonte de 2026 a 2030 será marcada pela convergência entre inteligência distribuída, arquiteturas orientadas a agentes e maior autonomia operacional. Nesse cenário, observa-se a transição de sistemas reativos e centralizados para ecossistemas capazes de perceber, decidir e agir de forma contínua, com mínima intervenção humana.

3.10.3.1. Tendências Estruturais

As principais tendências estruturais apontam para uma redefinição do paradigma de operação de redes, com destaque para os seguintes eixos:

- **Redes autônomas e ZSM**: a consolidação do *Zero-touch Network & Service Management* (ZSM) amplia a capacidade de auto-operação, autoconfiguração e automanutenção das redes. Nesse modelo, a atuação humana desloca-se para funções de supervisão, definição de políticas e governança, enquanto os sistemas executam ciclos fechados de controle [Lira et al. 2024, Elkael et al. 2026].
- **Sistemas multiagentes e Internet of Agents (IoA)**: a emergência de ecossistemas compostos por múltiplos agentes inteligentes apresenta novos desafios. Nesse contexto, entidades autônomas são capazes de descobrir serviços, negociar estratégias e cooperar dinamicamente por meio de protocolos específicos, como MCP e A2A [Kong et al. 2025, Elkael et al. 2026].
- **Integração com OpenRAN**: a incorporação de inteligência agentica ambientes OpenRAN, especialmente com as r/xApps, viabiliza o controle em malha fechada com granularidade temporal reduzida, aproximando a automação de requisitos operacionais em tempo real o que é essencial para diversas aplicações e serviços do ecossistema OpenRAN [Elkael et al. 2026].
- **Redes autoadaptativas e *self-healing***: a evolução para redes capazes de aprender

continuamente a partir de dados operacionais permite a identificação proativa de falhas, degradações e interferências, com geração automática de estratégias de mitigação. Iniciativas como AI-RAN Factory ilustram esse movimento em direção a sistemas autoevolutivos [Elkael et al. 2026, Boateng et al. 2024b].

3.10.3.2. Desdobramentos Práticos e Tecnológicos

A partir dessas tendências estruturais, emergem desdobramentos concretos que impactam diretamente a implementação e operação de sistemas de rede:

- **Arquiteturas *SLM-first***: observa-se a consolidação de arquiteturas heterogêneas nas quais *Small Language Models* (SLMs) atuam como padrão operacional para tarefas recorrentes, enquanto *Large Language Models* (LLMs) são acionados sob demanda para cenários que exigem maior capacidade de raciocínio. O ajuste fino de SLMs sobre dados organizacionais, como logs, *playbooks* e históricos de configuração, tende a proporcionar ganhos significativos em eficiência e custo [Microsoft Research 2025, Liquid AI 2025].
- **Interoperabilidade entre agentes**: a evolução de protocolos como MCP e A2A em direção a modelos de governança abertos favorece a interoperabilidade entre agentes e sistemas heterogêneos. Espera-se que controladores SDN, plataformas de monitoramento e orquestradores de NFV passem a expor interfaces nativas compatíveis, reduzindo a necessidade de integrações específicas e promovendo maior composição de serviços [Anthropic 2025, Google 2025].
- **Consolidação da *Edge AI***: a viabilidade de execução de modelos compactos em dispositivos com recursos limitados amplia o uso de agentes inteligentes em ambientes de borda, como IoT industrial e infraestruturas O-RAN. A inferência local passa a desempenhar papel central não apenas na redução de latência e custo, mas também na garantia de disponibilidade em cenários com conectividade intermitente.
- **Novos paradigmas de avaliação**: a avaliação de sistemas inteligentes desloca-se do desempenho isolado de modelos para a análise de sistemas completos capazes de tomar decisões e executar ações. *Benchmarks* recentes, como SWE-bench Verified e GAIA, evidenciam a importância de métricas relacionadas à qualidade do raciocínio intermediário, eficiência de planejamento e capacidade de autocorreção [SWE-bench Team 2025, Mialon et al. 2023].

3.10.4. Oportunidades de Pesquisa e Inovação

Apesar dos avanços recentes, diversas áreas ainda demandam investigação técnica e acadêmica aprofundada:

- **Interoperabilidade entre domínios**: desenvolvimento de arquiteturas de agentes capazes de operar de forma integrada em ambientes heterogêneos, incluindo segmentos satelitais, aéreos e terrestres [Boateng et al. 2024b, Du et al. 2026].
- **Eficiência de inferência em tempo real**: otimização de modelos e arquiteturas para atender aos requisitos de latência extremamente baixa previstos para redes 6G e sistemas críticos distribuídos [Boateng et al. 2024b, Chowa et al. 2026].

- **Segurança e responsabilização**: criação de mecanismos de auditoria forense, rastreabilidade de decisão e atribuição de responsabilidade em falhas causadas por agentes, além de proteção contra ataques de exaustão cognitiva e abuso de ferramentas [Kong et al. 2025, Bandara et al. 2025].
- **Padronização de *benchmarks***: estabelecimento de métricas robustas para avaliar inteligência, confiabilidade, custo operacional e segurança de agentes em cenários de rede realistas [Du et al. 2026, Chowa et al. 2026].

Em síntese, este minicurso demonstrou que a construção de agentes inteligentes constitui um passo decisivo para transformar redes em sistemas mais resilientes, adaptativos e capazes de ampliar continuamente sua inteligência operacional. Mais do que automatizar tarefas isoladas, a combinação entre modelos de linguagem, arquiteturas de agentes, RAG e mecanismos de verificação aponta para uma nova geração de infraestruturas capazes de aprender com a própria operação e responder de forma mais eficaz às demandas da próxima década [Elkael et al. 2026, Huang et al. 2023].

Agradecimentos: A elaboração deste minicurso faz parte de projetos apoiados pelo Ministério da Ciência, Tecnologia e Inovações, com recursos da Lei nº 8.248/1991, no âmbito do PPI-SOFTEX, coordenado pela Softex e publicado na Arquitetura Cognitiva (Fase 3), DOU 01245.003479/2024-10, bem como de projetos do INCT Redes de Comunicação e Internet das Coisas Inteligentes, financiados pelo CNPq (processo nº 405940/2022-0) e FAPESP (processo nº 2023/00673-7).

Referências

[Anthropic 2024] Anthropic (2024). Contextual retrieval. `https://www.anthropic.com/news/contextual-retrieval`.

[Anthropic 2025] Anthropic (2025). Model context protocol specification. modelcontextprotocol.io.

[Bai et al. 2022] Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., Chen, C., Olsson, C., Olah, C., Hernandez, D., Drain, D., Ganguli, D., Li, D., Tran-Johnson, E., Perez, E., Kerr, J., Mueller, J., Ladish, J., Landau, J., Ndousse, K., Lukosuite, K., Lovitt, L., Sellitto, M., Elhage, N., Schiefer, N., Mercado, N., DasSarma, N., Lasenby, R., Larson, R., Ringer, S., Johnston, S., Kravec, S., Showk, S. E., Fort, S., Lanham, T., Telleen-Lawton, T., Conerly, T., Henighan, T., Hume, T., Bowman, S. R., Hatfield-Dodds, Z., Mann, B., Amodei, D., Joseph, N., McCandlish, S., Brown, T., and Kaplan, J. (2022). Constitutional ai: Harmlessness from ai feedback.

[Bandara et al. 2025] Bandara, E., Gore, R., Foytik, P., Shetty, S., Mukkamala, R., Rahman, A., Liang, X., Bouk, S. H., Hass, A., Rajapakse, S., Keong, N. W., Zoysa, K. D., Withanage, A., and Loganathan, N. (2025). A practical guide for designing, developing, and deploying production-grade agentic ai workflows.

[Boateng et al. 2024a] Boateng, G. O. et al. (2024a). A survey on large language models for communication, network, and service management: Application insights, challenges, and future directions. *arXiv preprint arXiv:2412.19823*.

[Boateng et al. 2024b] Boateng, G. O., Sami, H., Alagha, A., Elmekki, H., Hammoud, A., Mizouni, R., Mourad, A., Otrok, H., Bentahar, J., Muhaidat, S., Talhi, C., Dziong, Z., and Guizani, M. (2024b). A survey on large language models for communication, network, and service management: Application insights, challenges, and future directions.

[Bommasani et al. 2022] Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N., Chen, A., Creel, K., Davis, J. Q., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M., Krishna, R., Kuditipudi, R., Kumar, A., Ladhak, F., Lee, M., Lee, T., Leskovec, J., Levent, I., Li, X. L., Li, X., Ma, T., Malik, A., Manning, C. D., Mirchandani, S., Mitchell, E., Munyikwa, Z., Nair, S., Narayan, A., Narayanan, D., Newman, B., Nie, A., Niebles, J. C., Nilforoshan, H., Nyarko, J., Ogut, G., Orr, L., Papadimitriou, I., Park, J. S., Piech, C., Portelance, E., Potts, C., Raghunathan, A., Reich, R., Ren, H., Rong, F., Roohani, Y., Ruiz, C., Ryan, J., Ré, C., Sadigh, D., Sagawa, S., Santhanam, K., Shih, A., Srinivasan, K., Tamkin, A., Taori, R., Thomas, A. W., Tramèr, F., Wang, R. E., Wang, W., Wu, B., Wu, J., Wu, Y., Xie, S. M., Yasunaga, M., You, J., Zaharia, M., Zhang, M., Zhang, T., Zhang, X., Zhang, Y., Zheng, L., Zhou, K., and Liang, P. (2022). On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

[Borgeaud et al. 2022] Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., van den Driessche, G., Lespiau, J.-B., Damoc, B., Clark, A., de Las Casas, D., Guy, A., Menick, J., Ring, R., Hennigan, T., Huang, S., Maggiore, L., Jones, C., Cassirer, A., Brock, A., Paganini, M., Irving, G., Vinyals, O., Osindero, S., Simonyan, K., Rae, J. W., Elsen, E., and Sifre, L. (2022). Improving language models by retrieving from trillions of tokens.

[Chakraborty et al. 2024] Chakraborty, S., Chitta, N., and Sundaresan, R. (2024). Automation of network configuration generation using large language models. In *Proc. 20th International Conference on Network and Service Management (CNSM)*.

[Chowa et al. 2026] Chowa, S. S., Alvi, R., Rahman, S. S., Rahman, M. A., Raiaan, M. A. K., Islam, M. R., Hussain, M., and Azam, S. (2026). From language to action: a review of large language models as autonomous agents and tool users. *Artificial Intelligence Review*, 59(2).

[de Lamo Castrillo et al. 2025] de Lamo Castrillo, V., Gidey, H. K., Lenz, A., and Knoll, A. (2025). Fundamentals of building autonomous llm agents.

[Derouiche et al. 2025] Derouiche, H., Brahmi, Z., and Mazeni, H. (2025). Agentic AI frameworks: Architectures, protocols, and design challenges. arXiv:2508.10146.

[Dettmers et al. 2023] Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms.

[Du et al. 2022] Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., Krikun, M., Zhou, Y., Yu, A. W., Firat, O., Zoph, B., Fedus, L., Bosma, M., Zhou, Z., Wang, T., Wang, Y. E., Webster, K., Pellat, M., Robinson, K., Meier-Hellstern, K., Duke, T., Dixon, L., Zhang, K., Le, Q. V., Wu, Y., Chen, Z., and Cui, C. (2022). Glam: Efficient scaling of language models with mixture-of-experts.

[Du et al. 2026] Du, S., Zhao, J., Shi, J., Xie, Z., Jiang, X., Bai, Y., and He, L. (2026). A survey on the optimization of large language model-based agents. *ACM Comput. Surv.*, 58(9).

[Elkael et al. 2026] Elkael, M., D'Oro, S., Bonati, L., Polese, M., Lee, Y., Furueda, K., and Melodia, T. (2026). Agentran: An agentic ai architecture for autonomous control of open 6g networks.

[Erdogan et al. 2024] Erdogan, L. E., Lee, N., Jha, S., Kim, S., Tabrizi, R., Moon, S., Hooper, C., Anumanchipalli, G., Keutzer, K., and Gholami, A. (2024). Tinyagent: Function calling at the edge.

[Errico et al. 2025] Errico, H., Ngiam, J., and Sojan, S. (2025). Securing the model context protocol (mcp): Risks, controls, and governance.

[Fedus et al. 2022] Fedus, W., Zoph, B., and Shazeer, N. (2022). Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. arXiv:2101.03961.

[Gao et al. 2023] Gao, Y. et al. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

[Google 2025] Google (2025). Agent-to-agent (A2A) protocol specification. Technical report, Google.

[Ha and Schmidhuber 2018] Ha, D. and Schmidhuber, J. (2018). World models. Zenodo. DOI: 10.5281/zenodo.1207631.

[Huang et al. 2023] Huang, Y., Du, H., Zhang, X., Niyato, D., Kang, J., Xiong, Z., Wang, S., and Huang, T. (2023). Large language models for networking: Applications, enabling techniques, and challenges.

[Jano 2024] Jano, P. W. (2024). Retrieval-augmented generation systems: A comprehensive survey of architectures, applications, and future directions. *University of Wisconsin–Madison Survey*.

[Jhandi et al. 2026] Jhandi, P., Kazi, O., Subramanian, S., and Sendas, N. (2026). Small language models for efficient agentic tool calling: Outperforming large models with targeted fine-tuning.

[Kaplan et al. 2020] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. arXiv:2001.08361.

[Karpukhin et al. 2020] Karpukhin, V., Oğuz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and tau Yih, W. (2020). Dense passage retrieval for open-domain question answering. arXiv:2004.04906.

[Khattab and Zaharia 2020] Khattab, O. and Zaharia, M. (2020). ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. arXiv:2004.12832.

[Kim 2026] Kim, M. H. (2026). Emergent cognitive convergence via implementation: Structured cognitive loop reflecting four theories of mind.

[Kong et al. 2025] Kong, D., Lin, S., Xu, Z., Wang, Z., Li, M., Li, Y., Zhang, Y., Peng, H., Chen, X., Sha, Z., Li, Y., Lin, C., Wang, X., Liu, X., Zhang, N., Chen, C., Wu, C., Khan, M. K., and Han, M. (2025). A survey of llm-driven ai agent communication: Protocols, security risks, and defense countermeasures.

[LangChain 2025] LangChain (2025). LangGraph: Building stateful multi-actor applications with LLMs. github.com/langchain-ai/langgraph.

[Lewis et al. 2020] Lewis, P. et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*.

[Lewis et al. 2021] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S., and Kiela, D. (2021). Retrieval-augmented generation for knowledge-intensive nlp tasks.

[Liquid AI 2025] Liquid AI (2025). LFM2.5: Liquid foundation models technical report. Technical report, Liquid AI.

[Lira et al. 2024] Lira, O. G., Caicedo, O. M., and da Fonseca, N. L. (2024). Large language models for zero touch network configuration management. *IEEE Communications Magazine*, 63(7):146–153.

[Liu et al. 2025] Liu, J., Kong, Z., Yang, C., Yang, F., Li, T., Dong, P., Nanjekye, J., Tang, H., Yuan, G., Niu, W., Zhang, W., Zhao, P., Lin, X., Huang, D., and Wang, Y. (2025). Rcr-router: Efficient role-aware context routing for multi-agent llm systems with structured memory.

[Long et al. 2025] Long, S., Tan, J., Mao, B., Tang, F., Li, Y., Zhao, M., and Kato, N. (2025). A survey on intelligent network operations and performance optimization based on large language models. *IEEE Communications Surveys & Tutorials*, 27(6):3915–3949.

[Madaan et al. 2023] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. (2023). Self-refine: Iterative refinement with self-feedback.

[Manning et al. 2008] Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

[Mialon et al. 2023] Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. (2023). Gaia: a benchmark for general ai assistants.

[Microsoft Research 2025] Microsoft Research (2025). Phi-4-mini technical report. Technical report, Microsoft.

[ML.ENERGY 2024] ML.ENERGY (2024). ML.ENERGY leaderboard: Energy consumption of large language models. ml.energy/leaderboard.

[NVIDIA Research 2025] NVIDIA Research (2025). Small language models are the future of agentic AI. arXiv:2510.03847.

[OpenAI 2025] OpenAI (2025). A practical guide to building agents. Technical report, OpenAI.

[Ouyang et al. 2022] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback.

[Pan et al. 2024] Pan, J. J., Wang, J., and Li, G. (2024). Survey of vector database management systems. *The VLDB Journal*.

[Park et al. 2023] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior.

[Rawat et al. 2025] Rawat, M., Gupta, A., Goomer, R., Bari, A. D., Gupta, N., and Pieraccini, R. (2025). Pre-act: Multi-step planning and reasoning improves acting in llm agents.

[Robertson and Zaragoza 2009] Robertson, S. and Zaragoza, H. (2009). The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*.

[Sanh et al. 2020] Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2020). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv:1910.01108.

[Schick et al. 2023] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools.

[Sharma and Mehta 2025] Sharma, R. and Mehta, M. (2025). Small language models for agentic systems: A survey of architectures, capabilities, and deployment trade offs.

[Singh et al. 2025] Singh, A., Ehtesham, A., Kumar, S., and Khoei, T. T. (2025). Agentic retrieval-augmented generation: A survey on agentic rag.

[SWE-bench Team 2025] SWE-bench Team (2025). SWE-bench verified leaderboard. swebench.com.

[Webb et al. 2025] Webb, T., Mondal, S. S., and Momennejad, I. (2025). Improving planning with large language models: A modular agentic architecture.

[Yao et al. 2023] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). React: Synergizing reasoning and acting in language models.

[Ye et al. 2025] Ye, R., Liu, X., Wu, Q., Pang, X., Yin, Z., Bai, L., and Chen, S. (2025). X-mas: Towards building multi-agent systems with heterogeneous llms.

[Yue et al. 2025] Yue, Y., Zhang, G., Liu, B., Wan, G., Wang, K., Cheng, D., and Qi, Y. (2025). Masrouter: Learning to route llms for multi-agent systems.

[Zhou et al. 2024] Zhou, H., Hu, C., Yuan, Y., Cui, Y., Jin, Y., Chen, C., Wu, H., Yuan, D., Jiang, L., Wu, D., Liu, X., Zhang, C., Wang, X., and Liu, J. (2024). Large language model (llm) for telecommunications: A comprehensive survey on principles, key techniques, and opportunities.

