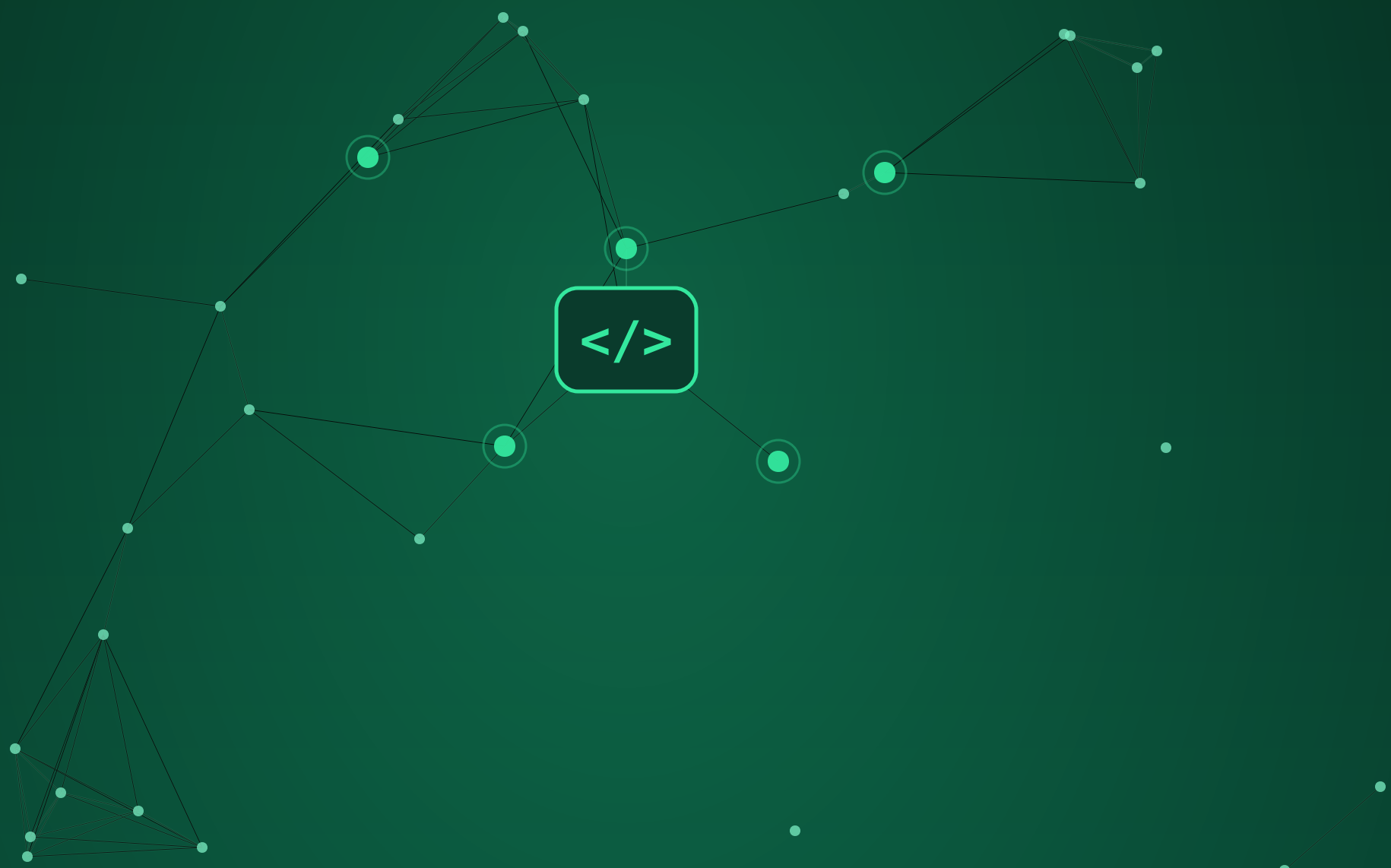

PRACTICAL AI · NETWORK ENGINEERING

AI for Network Engineers

A Practical Playbook for Automation, Troubleshooting, and Smarter Operations

19 chapters · 5 parts · runnable Python throughout
Cisco IOS / NX-OS · Juniper Junos · cloud & local models

JOZEF BAROŠ

PRACTICAL AI · NETWORK ENGINEERING

AI for Network Engineers

A Practical Playbook for Automation, Troubleshooting, and Smarter Operations

19 chapters · 5 parts · runnable Python throughout
Cisco IOS / NX-OS & Juniper Junos · cloud & local models

JOZEF BAROŠ

AI for Network Engineers — A Practical Playbook for Automation, Troubleshooting, and Smarter Operations.

Disclaimer. This book is provided for educational purposes. All code and techniques are supplied “as is,” without warranty of any kind. Always test scripts and configuration changes in a lab before using them in production, and apply changes only through your organisation's approved change-management process. The author accepts no liability for any loss or damage arising from the use of the material in this book.

Trademarks. Cisco, IOS, and NX-OS are trademarks of Cisco Systems, Inc. Juniper and Junos are trademarks of Juniper Networks, Inc. Other product, model, and company names are the trademarks or registered trademarks of their respective owners and are used for identification only. This book is independent and is not authorised, sponsored, or endorsed by any of these companies.

On AI-generated output. The models discussed in this book are probabilistic and can be confidently wrong. Every technique here assumes a human reviews AI output before it affects a network. Treat all generated configuration, commands, and conclusions as drafts to be verified — never as authority.

First edition, 2026.

Preface

Network engineers have always automated themselves out of repetitive work — from Expect scripts to Jinja2 templates to model-driven configuration. Large Language Models are the next step in that lineage, and this book is about using them as a serious working tool: not a chatbot novelty, but a co-worker that reads logs faster than you can, drafts configuration in your house style, and turns a wall of `show` output into a plain-English answer.

It is a deliberately practical book. Almost every section ends in code you can run, and the chapters build toward three complete projects you can deploy. The goal is not to teach you machine learning — it is to make you faster and sharper at the network job you already do, while never letting a probabilistic tool make an unsupervised change to your network.

Who this book is for

It is written for working network engineers — CCNP/CCIE-level operators, NetDevOps practitioners, and NOC teams — who are comfortable at the CLI and can read Python, but who are not AI specialists. If you have ever thought "AI could probably help with this, but I don't know where to start or whether it's safe," this book is for you.

What you will be able to do

- Run AI from Python against real network data, using either a cloud API or a local model with the same code.
- Summarise and triage logs, generate and validate configuration, and build a guided troubleshooting assistant.
- Ground every answer in real device data, get reliable structured output, and keep sensitive configs inside your perimeter.
- Build and safely deploy three end-to-end tools: a syslog triage bot, a fleet compliance checker, and a troubleshooting assistant.

The philosophy running through every chapter

Three ideas recur because they are what make AI safe and useful on a network. **Ground the model in real data** rather than letting it recall — paste the config, the log, the output. **The model proposes; a deterministic check disposes** — a diff, a dry-run, a linter, or a human approves anything that touches a device. And **a confident tone is never evidence of correctness** — always verify. Hold these and everything else in the book is detail.

How to read it

The five parts build in order: Part I lays the foundations and the reusable `chat()` client every later tool imports; Part II is the core use cases; Part III is the production-grade Python and patterns; Part IV is tools, local models, and governance; Part V assembles three complete projects. You can read straight through, or — once you have Part I — jump to whichever use case you need. Type the code out or adapt it to your own lab as you go; you learn this material by running it, not by reading it.

A note on examples: they deliberately mix Cisco IOS/NX-OS and Juniper Junos, because to a language model both are simply text and most real networks are mixed. Everything is written so you can point it at a cloud or a local model by changing a single line of configuration.

Table of Contents

Chapter 1 — Why AI Now: A Practical Introduction

Network engineering has always rewarded people who automate themselves out of repetitive work. We went from typing commands into a terminal, to scripting with Expect, to templating configurations with Jinja2, to declaring intent in YAML and letting tooling render the rest. Each step removed a layer of manual toil. Large Language Models (LLMs) — the technology behind tools like ChatGPT, Claude, and locally run models such as Llama — are the next layer, and this book is about using them as a working tool, not a novelty.

This is a hands-on book. Every concept is tied to something you can run: a script that summarises a log, a workflow that drafts a config, a pattern that turns a wall of `show` output into a plain-English answer. The goal is not to teach AI theory — it is to make you faster and sharper at the network job you already do.

This opening chapter gives you the foundation the rest of the book builds on: what genuinely changed with LLMs, an honest map of where AI helps and where it does not, the two ways you will run models throughout the book (cloud APIs and local models), and a safe, repeatable workflow. By the last page you will have made your first AI call from Python against a real network log.

Section	Focus	What you can do after it
1.1	From scripts to reasoning	Understand why LLMs differ from prior automation
1.2	Where AI helps — a use-case map	Judge instantly whether a task fits AI
1.3	Cloud APIs vs. local models	Run the same task two ways and choose well
1.4	Your first practical workflow	Safely explain a real device error with code

Examples throughout the book are multi-vendor: you will see both Cisco IOS/NX-OS and Juniper Junos, because to a language model both are simply text, and most real networks are mixed. Code is written so you can point it at a cloud or a local model by changing a couple of lines.

> 🛠️ **Pro Tip — How to read this book**
>
> Skim a section's headings first, then read the code with its caption, then read the prose around it. Every script in the book is self-contained and runnable — type it out or adapt it to your own lab as you go. You learn this material by running it, not by reading it.

1.1 From Scripts to Reasoning: What Actually Changed

A short history of removing toil

To understand why AI matters now, it helps to see it as the latest move in a long game. Every generation of network automation has answered the same question — *how do I stop doing this by hand?* — with a more capable tool.

- **Manual CLI.** You SSH into a device and type. Total control, zero scale: a change across 200 switches is 200 sessions.
- **Expect / shell scripts.** You record the keystrokes and replay them. This scales the typing but breaks the moment a prompt or banner changes.
- **Templating (Jinja2 + YAML).** You separate the *data* (hostnames, VLANs, IPs) from the *structure* (the config skeleton). One template renders hundreds of consistent configs. This is still the backbone of modern NetDevOps.
- **Intent / model-driven (NETCONF, RESTCONF, YANG).** You declare the desired state and let the device reconcile reality to match. Configuration becomes data you can validate against a schema.
- **Reasoning (LLMs).** You describe a goal or paste a problem in plain language, and the system *interprets* it — even if it has never seen that exact phrasing before.

Notice the trend: each layer handles more ambiguity than the last. Templating still demands perfectly structured data; NETCONF still demands a precise YANG model. LLMs are the first layer that tolerates messy, human, incomplete input — which is exactly what real network operations produce all day long.

Deterministic tools vs. probabilistic tools

Here is the single most important idea in the book, so we will define the terms precisely. A **deterministic** tool gives the same output for the same input every time. `ping 8.8.8.8` either reaches the host or it does not; a Jinja2 template renders byte-for-byte identically on every run. A **probabilistic** tool — which is what an LLM is — samples its answer from a distribution. Ask it the same question twice and you may get two differently worded, and occasionally differently correct, answers.

This is not a flaw to be fixed; it is the source of the model's flexibility. But it shapes how you use the tool. You never wire a probabilistic component directly to a destructive action. You treat its output as a *draft or a suggestion* that a deterministic check — a schema validation, a config diff, a dry-run, or a human — approves before anything touches the network. Keep that sentence in mind; the entire book is an elaboration of it.

Three terms you actually need

You can read this whole book knowing just three pieces of vocabulary:

- **Token.** The unit an LLM reads and writes — roughly ¾ of a word in English; a line of config is several tokens. Models have a maximum **context window** (e.g. 200,000 tokens): the total text they can consider at once. A giant `show tech-support` can exceed it, which is why later chapters teach you to filter before you send.

- **Inference.** A single run of the model that turns your input (the **prompt**) into output. Cloud inference happens on a provider's GPUs and is billed per token; local inference happens on your own hardware.
- **Hallucination.** When the model produces confident, fluent text that is simply false — an interface name that does not exist, a command flag never implemented. Managing this is a recurring theme, and §1.4 gives you the workflow that contains it.

> 💡 **Did you know? — LLMs do not 'look things up'**
>
> A base language model has no live connection to your network or the internet. It answers from statistical patterns learned during training. When you paste a log and it 'reads' it, that text is simply part of the prompt for that one inference — it is not stored, and the model has no memory of it next time unless you send it again. Chapter 12 (Retrieval-Augmented Generation) is the technique for giving the model trustworthy, current facts to reason over.

> 🌐 **Use Case — Why the draft mindset matters on a change window**
>
> A real pattern from production: an engineer asked a chatbot to 'generate the config to add VLAN 50 to all access ports' and pasted the output straight into a Catalyst at 2 a.m. The model helpfully included `switchport mode access` on a line that was actually an uplink trunk, collapsing the link to the distribution layer. The takeaway is not 'avoid AI' — it is that AI output is a draft, and a trunk port is a deterministic fact the model guessed wrong. A two-second `show run interface` diff before applying would have caught it. Throughout this book, AI proposes and a deterministic check disposes.

> **Key Takeaways**
>
> **LLMs reason over unstructured text** — exactly what operations produces. **They are probabilistic, not deterministic**: treat every output as a draft, never wire it straight to a destructive action. **A token is ~¾ of a word**; context windows are finite, so filter before you send. **Hallucination is the core risk**, contained by deterministic checks and review — never by trust.

1.2 Where AI Genuinely Helps — A Use-Case Map

If you over-promise, your first wrong answer destroys trust; if you under-use, you leave real time on the table. The honest position is that AI is excellent at a specific *shape* of task and poor at another. Learn the shape and you will know at a glance whether to reach for it.

The shape of a good AI task

AI excels when a task is **language-heavy, tolerant of a draft, and cheap to verify**. Summarising a log, explaining an error, drafting a config from a description, translating a Cisco config into its Junos equivalent — in all of these a human can glance at the result and immediately judge whether it is right. Verification is fast and the cost of a wrong draft is low because nothing is applied yet.

AI is a poor fit when a task demands **guaranteed correctness, real-time precision, or ground truth the model cannot see**. Computing the exact set of prefixes a route-map permits, deciding whether to fail traffic over during a live incident, or asserting the current state of an interface — these need deterministic tools (a real device, a parser, a route simulator), not a probabilistic guess.

A practical task map worth keeping at your desk

Task	Fit	Why
Summarise a 5,000-line log	High	Language-heavy, draft is fine, easy to spot-check
Explain a cryptic commit/error	High	Interpretation of text it can pattern-match
Draft a config from a description	High	Output is reviewed and diffed before apply
Translate IOS ↔ Junos syntax	High	Structured language mapping, verifiable
Write a parsing/automation script	High	You run and test it deterministically
Generate documentation from configs	High	Tedious, low-risk, human edits after
Decide live failover during an incident	Low	Needs ground truth and guaranteed logic
Compute exact route-map / ACL outcome	Low	Use a simulator or the device, not a guess
Assert current interface / BGP state	Low	The model cannot see live state

The use cases this book will build

The rest of the book is organised around concrete, runnable use cases. Each one is a self-contained tool you can adapt to your own network:

1. **Log and syslog analysis** — feed thousands of lines, get a ranked summary of what matters (Chapter 6).
2. **Config generation and validation** — turn an intent into a config, then check a config against your standard (Chapter 5).
3. **A troubleshooting co-pilot** — symptom in, hypotheses and diagnostic commands out, grounded in real device output (Chapter 7).

4. **Documentation automation** — generate diagrams, runbooks, and as-built docs straight from configs (Chapter 8).
5. **Ask-your-network** — a private chatbot over your own configs and standards using retrieval (Chapter 12).
6. **Automation agents** — scripts that fetch data, reason, and propose the next step, with a human gate (Chapter 13).

A worked contrast

Consider one syslog line on a Cisco device:

A line you might see a hundred times and still not recall

```
%BGP-5-NBRCHANGE: neighbor 10.0.0.2 Up
```

Asking AI to *explain* this — what BGP severity 5 means, what `NBRCHANGE` indicates, what to check next — is a high-fit task: it is interpretation, the answer is verifiable, and nothing is changed. But asking AI *"is my BGP session to 10.0.0.2 currently up?"* is a low-fit task: the only source of truth is the device itself, which the model cannot see. The reliable fix is to let a deterministic tool fetch the live state and hand it to the model to interpret — the exact pattern you build in Chapters 6 and 11. In short: never ask the model for facts it cannot observe; fetch them and let it reason over real data.

> 🌐 **Use Case — The 20-minute task that became 2 minutes**
>
> A common, genuinely safe win: after every maintenance window, an engineer had to read the night's syslog and write a human summary for the morning handover. AI drafts that summary in seconds from the pasted log; the engineer edits and signs it. The task did not become unsupervised — it became faster. That is the realistic promise of this book: not autonomy, but acceleration with a person in the loop.

> **Key Takeaways**
>
> **Good AI tasks are language-heavy, draft-tolerant, and easy to verify. Poor AI tasks need guaranteed correctness or live state** the model cannot see. **Never ask the model for facts about your network** — fetch them and hand them over. **The book is a sequence of runnable use cases**, each a tool you adapt to your own environment.

1.3 The Two Engines: Cloud APIs and Local Models

Throughout this book you run AI in two ways, and choosing the right one per task is a core skill. A **cloud API** sends your prompt over HTTPS to a provider (OpenAI, Anthropic, Google) whose large models run on their GPUs; you pay per token and get state-of-the-art quality with no hardware to manage. A **local model** runs entirely on your own machine or server — commonly via Ollama or LM Studio — so your data never leaves your control and inference is free after the hardware cost, at the price of needing a capable machine and accepting somewhat lower quality from smaller models.

The trade-off, at a glance

Dimension	Cloud API	Local model
Quality	Highest (frontier models)	Good; best on focused tasks
Data privacy	Leaves your network — review policy	Stays on your hardware
Cost model	Per token (pennies, but adds up)	Free per call after hardware
Latency	Network round-trip	Depends on your GPU / RAM
Offline / air-gapped	No	Yes — works fully offline
Setup effort	API key, minutes	Install runtime, pull a model

There is no single winner. The decision usually comes down to one question: **may this data leave the building?** A public release note or a sanitised lab log can go to a cloud API for top quality. A production running-config full of addressing, secrets, and customer detail often must stay local — or be redacted first (a technique in Chapter 16). Many mature teams run *both*: a cloud model for general and code tasks, a local model behind the firewall for anything touching customer data.

> 🌐 **Use Case — Same code, different endpoint**
>
> In a regulated telco it is common to run an internal LLM proxy that routes general questions to a frontier cloud model while keeping config- and customer-data prompts on an on-premises model — the application code is identical, only the endpoint URL changes. Learning both engines means you can place each task on the right side of the privacy line instead of being forced into one tool. The scripts in this book are written exactly this way.

Your first cloud API call

Every cloud provider exposes the same essential shape: you send a list of messages and receive a generated reply. Here is a minimal, real call that asks a frontier model to explain a Cisco log line. The API key is read from an environment variable — never hard-code secrets, a rule Chapter 16 makes non-negotiable.

first_call.py — explain a Cisco syslog line with a cloud model

```
import os
from anthropic import Anthropic   # pip install anthropic

client = Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"])
```

```
log_line = "%BGP-5-NBRCHANGE: neighbor 10.0.0.2 Down BGP Notification sent"

resp = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=400,
    system="You are a senior network engineer. Be concise and precise.",
    messages=[{
        "role": "user",
        "content": f"Explain this Cisco IOS syslog line and list 3 things to
check:\n{log_line}",
    }],
)
print(resp.content[0].text)
```

The OpenAI SDK is nearly identical in spirit — a `client.chat.completions.create(...)` call with the same idea of a system message plus user messages. Once you have written one, you can read any of them. Chapter 10 covers retries, timeouts, streaming, and counting tokens for cost control.

The same task, run locally

Now the local equivalent. After installing Ollama and pulling a model once (`ollama pull llama3.1`), the model runs on your machine and the prompt never leaves it — ideal for a real running-config. The interface is again just an HTTP call, so the mental model carries straight over.

local_call.py — same explanation, fully offline via Ollama

```
import requests   # pip install requests

log_line = "%BGP-5-NBRCHANGE: neighbor 10.0.0.2 Down BGP Notification sent"

resp = requests.post("http://localhost:11434/api/chat", json={
    "model": "llama3.1",
    "stream": False,
    "messages": [
        {"role": "system", "content": "You are a senior network engineer. Be
concise."},
        {"role": "user",
         "content": f"Explain this Cisco IOS syslog line and list 3 things to
check:\n{log_line}"},
    ],
})
print(resp.json()["message"]["content"])
```

These two scripts do the *same job* with the *same structure* — a system instruction, a user message, a generated reply. The only real differences are where the model runs and who can see the data. That symmetry is deliberate: most of this book's code is written so you can switch engines by changing a few lines, which is exactly how you keep sensitive Junos and IOS configs in-house while still using a cloud model for everything else.

> 💡 **Did you know? — Junos and IOS both travel as plain text**

Whether you paste a Cisco `show ip bgp summary` or a Juniper `show bgp summary`, to the model it is just text in the prompt. That is why a single workflow can serve a multi-vendor network: the engine does not care about the platform, only the tokens. The vendor-specific skill lives in *your* prompt and *your* verification, not in the model's plumbing.

🛠️ **Pro Tip — Start cloud, graduate to local where needed**

If you are new to this, begin with a cloud API in a lab using only non-sensitive data — you get the best quality with the least setup and learn the patterns fastest. Introduce a local model when a real task requires data that must not leave your network. Standing up GPU infrastructure on day one is the quickest way to stall a promising automation effort.

Key Takeaways

Cloud = top quality, trivial setup, data leaves, per-token cost. Local = data stays, free per call, offline-capable, needs hardware, smaller models. The deciding question is 'may this data leave the building?' The same code shape works for both engines — swap the endpoint. **Never hard-code API keys**; read them from the environment.

1.4 Your First Practical Workflow

Everything in this book gets safer and more productive if you hold one analogy in your head. Treat the AI as **a brilliant, tireless, but unverified junior engineer.** It reads faster than anyone on your team, it has seen more configs and logs than any human ever will, and it never gets bored at 3 a.m. It is also occasionally and confidently wrong, has never seen *your* network, and must never be handed the enable password unsupervised.

Three rules that keep you safe

1. **Read before write.** Begin with tasks where AI only *reads and explains* — summarising logs, interpreting errors, drafting documentation. Earn trust there before letting it draft anything that will be applied. Most of this book's value lives on the read path, and it carries almost no risk.
2. **Draft, then verify deterministically.** When AI produces something that will touch the network — a config snippet, a script — a deterministic gate must approve it: a config diff, a `commit confirmed` on Junos, a dry-run, a linter, or a human review. The model proposes; a deterministic check disposes.
3. **Ground it in real data.** Never ask the model to recall facts about your network. Fetch the live data with a tool and hand it over. A grounded model interpreting real output is reliable; an ungrounded model guessing is dangerous. And remember: a confident tone is never evidence of correctness — always verify, because the cost of verification is tiny next to the cost of an outage.

> 🌐 **Use Case — Why 'junior engineer' is the right frame**
>
> You would happily let a sharp junior draft a change, summarise an incident, or write a first-pass script — and you would review their work before it shipped. You would not hand them the keys to the core routers on their first night and walk away. Applying exactly that posture to AI gives you most of the upside with little of the risk, and it scales: as your verification tooling matures, you can safely delegate more.

Your first contact: explain a real device error

Let us make the read path concrete. Below is a Junos commit error that has cost many engineers ten minutes of squinting. We will ground the model in the *actual* error text and ask it to interpret — the safest, highest-value pattern in the whole book.

explain_commit_error.py — grounded interpretation of a real Junos error

```
import os
from anthropic import Anthropic

client = Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"])

# Real text fetched from the device — we GROUND the model in it.
commit_error = '''
[edit interfaces ge-0/0/3 unit 0 family inet]
  'address 10.1.1.1/24'
    Cannot assign address; already in use on ge-0/0/1.0
```

```
error: configuration check-out failed
'''

prompt = (
    "You are a Juniper Junos expert. A junior engineer hit this commit "
    "error. Explain in plain English what it means, why it happened, and "
    "the exact steps to resolve it. Do not invent commands you are unsure of.\n\n"
    f"{commit_error}"
)

resp = client.messages.create(
    model="claude-sonnet-4-6", max_tokens=500,
    messages=[{"role": "user", "content": prompt}],
)
print(resp.content[0].text)
```

Two choices in that prompt are worth naming, because you will reuse them constantly. First, we **assigned a role** ("a Juniper Junos expert"), which reliably sharpens the answer. Second, we **explicitly told it not to invent commands** — a simple instruction that measurably reduces hallucination on technical tasks. These are your first two prompt-engineering moves; Chapter 4 builds a whole toolkit on top of them.

> 🛠️ **Pro Tip — Add an escape hatch to risky prompts**
>
> For any prompt where a wrong answer could mislead a change, append a line like: 'If you are not certain, say so and state what you would verify on the device first.' Giving the model explicit permission to be unsure converts some hallucinations into honest 'I'd check X' answers — exactly what a good junior engineer would say.

> ▶️ **Try It Yourself — Run the read path in your own lab**
>
> Before moving on, try these three grounded, zero-risk tasks against real text from a lab device. Each one only reads and explains, so nothing can break:

1. Paste the output of `show logging` (IOS) or `show log messages` (Junos) and ask the model to summarise the five most important events.
2. Paste a config snippet and ask it to explain, line by line, what each command does — then verify two lines against the vendor docs.
3. Paste any error you do not recognise and ask for the likely cause and the exact commands to confirm it — then run those commands and compare.

Together these build the instinct for what a grounded prompt feels like and where the model is strong. That instinct is the real prerequisite for everything that follows.

> **Key Takeaways**
>
> **Model the AI as an unverified junior engineer. Rule 1: read before write. Rule 2: draft, then verify with a deterministic gate. Rule 3: ground it in real fetched data, never recall — and verify, because confidence is not correctness. Assign a role and forbid invention** in your prompts to cut hallucination. **Practise on the read path first** — it is safe and it builds judgement.

SNEAK PEEK

Chapter 6

Log & Syslog Analysis

The use case that sells AI to a sceptical engineer in five minutes — turning a wall of syslog into a ranked, plain-English picture of what actually matters.

The next two pages are the opening of Chapter 6.
The full chapter — and 18 others — are in the complete book.

Chapter 6 — Log and Syslog Analysis

If there is one task that sells AI to a sceptical network engineer in five minutes, it is log analysis. Logs are voluminous, repetitive, and exactly the kind of unstructured text the model reads better than any regex you could write under pressure. This chapter turns a wall of syslog into a ranked, plain-English picture of what actually matters — and it does so entirely on the safe read path, since reading logs never changes a device.

The chapter has one cardinal rule, stated up front because every section depends on it: **filter first, then summarise.** You will learn to slice a log down to the relevant lines before sending, to triage and rank events into structured data, to draw root-cause hints from correlated events, and finally to assemble a small syslog triage bot that runs the whole flow and posts a summary to your chat tool.

Section	Focus	What you build
6.1	Filter first, then summarise	A pre-filter that shrinks logs 90%
6.2	Triage & severity ranking	A ranked, structured event list
6.3	Root-cause hints	Correlated multi-source analysis
6.4	A syslog triage bot	An end-to-end notify pipeline

> 🛠️ **Pro Tip — Logs are the perfect first project**
>
> Because reading a log changes nothing, log analysis is the ideal place to build trust in AI tooling. Start here, prove the value with zero risk, and you earn the credibility to tackle the write-path chapters later.

6.1 Filter First, Then Summarise

The instinct to paste a whole 50,000-line log into a model is the single biggest mistake in this area, for the reasons in Chapter 2: you blow the token budget, you pay for thousands of irrelevant lines, and — worst — the relevant line may fall outside the context window and be silently ignored. The fix is deterministic pre-filtering. Use the tools you already trust (`grep`, `include`, Python) to extract the relevant slice, *then* hand a few hundred focused lines to the model.

prefilter.py — shrink a huge log to the relevant window before sending

```
import re
from llm import chat

def prefilter(path, around=None, severities=("%.*-0-", "%.*-1-",
                                             "%.*-2-", "%.*-3-")):
    """Keep only high-severity lines, plus context around a keyword."""
    keep = []
    sev_re = re.compile("|".join(severities))
    for line in open(path, errors="ignore"):
        if sev_re.search(line) or (around and around in line):
            keep.append(line.rstrip())
    return keep

lines = prefilter("core.log", around="GigabitEthernet0/1")
print(f"Filtered to {len(lines)} lines")

summary = chat(
    "Summarise these filtered syslog lines for a shift handover. "
    "Group by severity; list the key events with timestamps. "
    "Use ONLY lines shown.\n\n" + "\n".join(lines[:400]),
    temperature=0,
)
print(summary)
```

The Cisco severity convention helps here: the digit in `%FACILITY-SEVERITY-MNEMONIC` runs 0 (emergency) to 7 (debug), so filtering to 0–3 keeps the lines that actually warrant attention and drops the routine chatter. You have turned 50,000 lines into a few hundred that fit comfortably and cost almost nothing — and the answer is *better*, because the model is not distracted by noise.

> 💡 **Did you know? — Smaller input often gives a better answer**
>
> It is counter-intuitive, but feeding the model less can improve accuracy. A focused 300-line slice lets it reason about the actual event; a 50,000-line dump buries that event in noise and may push it out of context entirely. Pre-filtering is not just a cost optimisation — it is an accuracy optimisation.

> 🌐 **Use Case — The crash buried on line 2**
>
> Recall Chapter 2's silently-truncated log: a 40,000-line capture where the triggering interface reset sat on line 2, outside the window, so the model answered from the tail and got it wrong. A pre-filter keyed on the interface name and severity would have surfaced that line 2 immediately and dropped the irrelevant remainder. The same model that failed on the raw dump succeeds on the filtered slice — the

Enjoying the sample?

Get the complete book

AI for Network Engineers

A Practical Playbook for Automation, Troubleshooting, and Smarter Operations

19 chapters · 5 parts · ~165 pages of runnable, practical content

Inside the full book:

- **Foundations** — how LLMs work, your AI toolkit, and prompt engineering for network tasks.
- **Core use cases** — config generation & validation, log analysis, a troubleshooting co-pilot, docs automation, and design review.
- **Building with Python** — robust API calls, parsing network data, RAG over your own configs, and bounded agents.
- **Tools & operations** — the tooling landscape, self-hosted local models, and security, privacy & cost control.
- **Three complete projects** — a syslog triage bot, a fleet compliance checker, and a troubleshooting assistant.

Cisco IOS / NX-OS & Juniper Junos · cloud & local models · every example runnable

Get the full book at

www.leanpub.com/ai-network

by Jozef Baroš