ChatGPT

Project Design & Rollout Plan: Network Automation SRE Portfolio

North Star & Objectives

Guiding Principle: The project's "north star" is to maximize learning, demonstrate technical competency, and share knowledge above all. This means every design choice and feature should serve as a learning opportunity, showcase mastery of relevant skills, and be well-documented to educate others. The end goal is a portfolio project so polished and robust that a recruiter or engineer would say, "We could use this," or "We should hire the person who built this." By targeting **Network Automation Engineer** and **SRE** roles, the project will emphasize both network-centric automation and site reliability engineering best practices.

Key Objectives:
- **Demonstrate Breadth and Depth:** Show proficiency across the full stack (frontend + backend) as well as DevOps tooling, networking, and automation. Use modern, in-demand technologies (per industry surveys) and even incorporate multiple languages or frameworks if it adds value [1].
- **Learn & Document:** Treat the project as a learning journal. Try new tools or techniques and document the journey. For example, if you integrate a new network automation library or set up a CI/CD pipeline for the first time, document the rationale and process. This not only reinforces your learning but also **shares knowledge** with anyone reading your portfolio.
- **Polish & Best Practices:** Adhere to software engineering best practices at a level above typical personal projects. This includes clean code and architecture, comprehensive testing, clear documentation, and intuitive UX/UI. The project should feel like a mini professional product. As one guide puts it, *"Impress recruiters with well-structured, readable code" by using proper commenting, formatting, and robust architecture choices* [2]. In short, **treat this like a production-ready application**, not a toy app.

Aligning with Target Roles

Network Automation Engineer Focus: For this role, emphasize features that automate network tasks and demonstrate knowledge of networking concepts. Potential elements to highlight: - **Automation Scripts/ Services:** Include functionality to automate network configurations or tests (e.g. pushing configs to a router, verifying connectivity, or retrieving device statuses via APIs). Use industry-standard tools/libraries where possible (for example, Python libraries like Netmiko/NAPALM for device automation, or Ansible playbooks for network tasks).
- **Networking Concepts:** Show you understand networking by incorporating relevant data. For instance, if the project manages devices, include concepts like IP addresses, VLANs, routing status, etc. If it's a monitoring tool, display real or simulated network metrics (latency, packet loss).
- **APIs and Protocols:** Implement or consume network APIs. This could mean using REST APIs of cloud networking services (AWS, GCP) or device APIs (NETCONF/RESTCONF) to fetch or apply configurations. Even a mock integration with something like a Cisco DevNet sandbox or OpenConfig data would be impressive.
- **Example Inspiration:** Consider the design philosophy of professional tools like NetBox and Nautobot, which are open-source network "source of truth" platforms. *NetBox, for example, became a go-to solution for*

modeling networks by providing a single, robust UI and programmable APIs for everything from cable maps to device configs [3]. While your project will be smaller in scope, aiming for that level of cohesive design (UI + API) and thorough data modeling is a worthy goal.

Site Reliability Engineer (SRE) Focus: SRE skills overlap with DevOps, emphasizing reliability, automation, and infrastructure. To target SRE:
- **Infrastructure as Code & Automation:** Show you can automate environments. You might include Terraform scripts or Ansible playbooks to set up the project's infrastructure (e.g., provisioning servers or cloud resources needed by your app). This demonstrates familiarity with cloud and provisioning, which SREs do frequently.
- **Monitoring & Metrics:** Build in observability. Expose metrics from your application (e.g., request throughput, error rates, response times) and aggregate them for display. You could integrate Prometheus for metrics collection and Grafana for a dashboard, or even a simple custom dashboard within your app for monitoring service health. The key is to show you understand how to measure and monitor a system's reliability.
- **Reliability Features:** Implement features that improve reliability: retry mechanisms for failed network calls, circuit breaker patterns to handle outages, graceful degradation of features if dependent services fail, etc. Even simulating failure scenarios (and documenting how the system copes) will highlight SRE mindset.
- **CI/CD & Testing:** SREs value continuous deployment with safety. Include a CI/CD pipeline that runs tests on each commit and perhaps automates deployment (more on this in DevOps section). Show testing not as an afterthought but as a core part of reliability (e.g., if the project manages network configs, maybe include tests that validate config syntax or simulate network conditions).
- **SRE Culture:** Optionally, define **SLIs/SLOs** for your application – for example, an SLO that the web UI will have 99% uptime or an API response time under X ms – and discuss how you'd meet and measure these. It's even more impressive if you integrate a way to track these. Advanced practitioners sometimes also experiment with **chaos engineering** to prove resilience. *For instance, advanced DevOps projects incorporate SRE practices like defining SLIs/SLOs and even chaos engineering to test system resilience* [4]. While not mandatory, mentioning or planning these shows you're aiming "a cut above the rest."

By aligning features with these roles, you ensure that **every part of the project is relevant**. Recruiters will see that not only have you built something cool, but it directly demonstrates the skills needed for the jobs you want – a point often emphasized in portfolio advice (*"Tailor to the Job: Align your portfolio to the specific role… highlight skills they emphasize."* [5]).

Benchmarks & Inspiration

Before diving into design, it's wise to study some exemplary projects and tools for inspiration:

- **Open-Source "Learning" Projects:** One developer-created open-source app (for DevOps learners) provides a great benchmark. It was built as a microservices stack with **multiple modern languages (Node.js, Python, Go, Java Spring Boot)** and databases (MongoDB, Postgres) to mirror real-world polyglot environments [1]. Notable features included an iterative build approach (start with a simple frontend, then progressively add microservices) and built-in educational features like a *system info page* (showing the runtime environment, container info, etc.) and a *service dashboard* to monitor if backend services are up [6]. The project also didn't shy away from the nitty-gritty: it had **unit and integration tests** (often missing in toy apps) and was **well-documented with explanations** of architecture decisions [6] [7]. Perhaps most impressively, it allowed running in a simplified mode

(using SQLite and JSON files instead of full DBs) to lower the barrier to entry, while still allowing migration to real databases later [8]. This focus on versatility, documentation, and testing is exactly what will make your project stand out. We can take a cue from this: use a modern tech stack, build iteratively with quick wins, include tests early, and document the "why" behind technical choices.

- **Professional Open-Source Tools:** Look at widely-used network or DevOps tools for quality standards. For example, **NetBox** (DigitalOcean's network source-of-truth) and its fork **Nautobot** are known for their strong documentation, plugin systems, and API-first design. They have clean, responsive web UIs coupled with comprehensive REST APIs, and they follow best practices like proper versioning, changelogs, and contributor guides. *NetBox's success is attributed to focusing on its core goal (network source of truth) with a thorough data model, rather than being an "all-in-one" with half-baked features* [3] [9]. Likewise, DevOps tools like **StackStorm** (event-driven automation) or **Rundeck** (runbook automation) could inspire features for job scheduling or event handling in your project. While your project scope is smaller, emulating the **level of polish** these tools have – e.g. a dedicated docs site, a help wiki, modular code, and even things like a contribution guide and unit tests – will signal professionalism.

- **Top Portfolios & Case Studies:** It can also help to see how other engineers present their projects. Some open-source portfolio projects or "awesome" lists can guide on structure. For instance, browsing GitHub's **Awesome Network Automation** list (networktocode) or **DevOps project collections** can spark ideas for features and best practices. Additionally, portfolio case studies on blogs (Dev.to, Medium) often stress storytelling and impact. Remember that it's not just *what* you build, but *how you articulate it*. As a blog on recruiter perspectives notes: don't just list features – **explain the project's purpose, challenges you overcame, and solutions you implemented to show your thought process** [10]. Weave this storytelling into your documentation or even the UI (for example, a "About this project" page).

By benchmarking against both individual open-source projects and industry-grade tools, you set a high bar for yourself. The idea is to **learn from the best**: adopt their good practices (architecture patterns, testing, docs) and avoid common pitfalls (lack of tests, outdated tech, poor onboarding). Every time you implement a feature, ask "How would a top engineer do this?" or "How would this be done in a production system?" This mindset will keep the project quality high.

System Design Overview

Now let's outline the design of the project itself. We'll design a system that is both **feature-rich** (to cover the skills we want to show) and **modular** (to allow iterative development and easy understanding of components). Here's a high-level view:

- **Project Concept:** A web-based Network Automation and Monitoring Dashboard (working title). This application will allow users (or just you, in a demo context) to manage and observe some network-related tasks through a friendly UI. For example, it could enable running automation scripts on network devices (like retrieving interface statuses, applying config templates), and display live status or metrics of those devices. Think of it as a mini "NetOps/SRE toolkit" – combining a control panel for automation with a monitoring view for reliability. This concept lets you show off both network scripting and reliability engineering. (Feel free to adjust the concept based on your interests – the key is it involves both **networking** and **reliability**.)

- **Architecture:** The system will be **full-stack**, composed of a **frontend** and one or more **backend** services, plus supporting components:

- **Frontend:** A single-page web application (SPA) built with a modern framework (e.g. **React** with TypeScript, or Vue/Angular if you prefer). React is a safe bet due to popularity. The UI will provide dashboards, forms, and visualization of data. This will communicate with backend via REST (or GraphQL) API calls. Aim for a clean, professional design (more on UI/UX later).
- **Backend:** Design an API server that follows RESTful principles. You can use **Python (Flask/FastAPI or Django)** or **Node.js (Express or NestJS)**, whichever you're more comfortable with or want to learn. Python is common in network automation (good for using libraries like Netmiko), whereas Node might handle concurrent tasks well; even Go could be considered for its performance and static typing (learning Go could be a plus for SRE work). It's perfectly fine to start with one backend service (monolith style), but structure it in modules (e.g., a module for "network device management", one for "monitoring data", etc.). This keeps the design modular and open to breaking into microservices later if desired.
    - If ambitious, you can adopt a **microservices** approach from the start: e.g., have one service written in Python that handles network device communications (SSH/API calls to routers), and another service (maybe in Node or Go) that handles serving the web API and orchestrating tasks. They could communicate over REST or a message queue. This multi-service, multi-language approach truly showcases polyglot skills and system design, just as the example project did with Node/Python/Go mix [1]. *However, note:* microservices add complexity (deployment, communication, etc.), so weigh this choice. Even as a single service, you can structure the code well to simulate separation of concerns.
- **Database:** Include a database to store persistent data. This might include things like: an inventory of network devices (with their IP, credentials securely stored, etc.), logs of automation actions performed, user accounts (if implementing auth), or metrics history for monitoring. A **SQL database** like PostgreSQL is a solid choice (especially since many infra tools use it, e.g. NetBox uses Postgres). Using an ORM (SQLAlchemy for Python/Flask or Django's ORM, or TypeORM/Prisma for Node) will help manage schema. Alternatively, a NoSQL DB like MongoDB could be used if the data is document-oriented – but SQL will showcase you know relational concepts which are still very relevant. You could even use both (one SQL, one NoSQL) if you want to show you can handle polyglot persistence, but that's optional.
    - As a simpler start, you might use **Sqlite** for initial development (no external dependency) and then migrate to Postgres for deployment. This is a tactic the DevOps learning project used to lower setup complexity [8].
- **Network Integration Layer:** This is a conceptual part of the backend responsible for actually talking to network devices or simulations. If you have access to real or virtual network devices (e.g., Cisco DevNet sandboxes, GNS3 simulations, or even dummy routers), this layer would contain the scripts or API calls. For example, it might use Netmiko to SSH into a device and run commands, or use a REST API if the device supports it. If you don't have actual devices, you can **simulate** this layer: create dummy functions that generate sample output (like "interface is up, traffic = X Mbps") or use a network simulator library like **Mininet** for a virtual network. The goal is to show *the capability* for network automation, even if it's on dummy data – you can explain in docs that "in a real scenario, this would integrate with actual network devices via library X".
- **Monitoring/Telemetry Module:** To highlight SRE, incorporate a subsystem for metrics and logs. For metrics, you could use something like **Prometheus** client libraries to collect app metrics (if using Python or Node, there are client libs that expose an endpoint like `/metrics` for Prometheus to

scrape). For logs, using a structured logging library and perhaps feeding logs to an external system (or at least storing them) would be good. A stretch goal: deploy a Grafana instance to visualize metrics, or integrate a small Grafana panel into your UI via iframe or API – or simply fetch recorded metrics and display as charts in your React app. At minimum, have an page in the frontend that shows system health (e.g., "All systems operational" with green lights for each service, like the service dashboard idea [6]).

- **External APIs/Services:** Consider if your project can leverage any external API for learning. For instance, maybe use a cloud API (AWS SDK or GCP) to show cloud networking automation (like creating a VPC or security group via code). Or use third-party services for some features: e.g., use an API for device geo-location to plot where devices are on a map (if that makes sense). This isn't required, but one portfolio tip is to *mention specific APIs/libraries used to show your range* [11]. So if you have any interesting integration (say, pulling data from a network performance API), it's a plus.

Key Features to Implement: (these will be spread across front and back ends)
- *Device Inventory:* A page to add/view network devices (even if virtual). Fields might include name, IP, type, status, etc. Backend provides CRUD for devices, perhaps storing them in the DB. This demonstrates basic full-stack CRUD functionality and use of a database.
- *Automation Actions:* Ability to perform actions on a device or network. For example, a "Run Diagnostics" button that triggers a backend job to fetch interface statuses or ping the device, then returns the results to display on UI. This can demonstrate asynchronous job handling (you might implement a job queue with something like Celery for Python or BullMQ for Node, or simpler threads/async tasks). Even if the action is just hitting a dummy function, implement it as if it were real (so you can show the pattern of background jobs, progress updates, etc.).
- *Monitoring Dashboard:* A section of the UI that displays metrics (maybe a graph of response times, or CPU usage of devices if you simulate that, or simply the uptime of your services). You can generate dummy metrics if needed. The main point is to show **data visualization** and concern for reliability. If using Prometheus/Grafana, you could embed a Grafana chart or use a chart library (like Chart.js or D3) to plot data from an API.
- *User Management (Optional):* If you want to show security/auth skills, include a simple authentication system. For example, a login for the web interface (with hashed passwords, JWT tokens for the API). This is relevant because many internal tools still have auth, and it demonstrates knowledge of security best practices. However, if time is short, this can be skipped or added later since it's not directly network/SRE specific.
- *Settings/Config Management:* Perhaps allow the user to tweak certain parameters (like how often to refresh data, or store config templates for network devices). This shows the ability to make the app flexible and adds depth.
- *Knowledge Sharing Integration:* To emphasize "sharing knowledge," consider incorporating a documentation or tutorial section *within* the project. For example, have an **About/Docs page** in the frontend that actually displays the project's README or a subset of your documentation, so a recruiter can read about the design decisions right inside the app. You could also provide tooltips or info icons in the UI explaining technical details (teaching the user). Essentially, bake documentation and educational aspects into the application's user experience.

The design above ensures you'll touch all areas: UI/UX, API development, database, external integrations, DevOps (with the environment setup, metrics, etc.), and of course network automation logic. It's a **broad yet cohesive project**: broad in the sense of technologies involved, cohesive in that all pieces serve the central theme of network automation and reliability.

Frontend Design & UX/UI Considerations

The frontend is the first thing anyone (recruiter or user) will see, so it should be polished and professional:

- **Tech Stack:** Use a popular framework (React is recommended). Set up a proper project structure (components, services for API calls, state management if needed – perhaps Redux or React Context for global state like user info or global notifications). Use TypeScript for type safety if you're comfortable; it's a plus for showing engineering discipline.
- **Design and Layout:** Aim for a clean, dashboard-style interface. You might use a UI library like **Material-UI (MUI)**, **Ant Design**, or **Chakra UI** to get a professional look with minimal custom CSS. These libraries come with pre-built components (navbars, tables, forms, modals) that can save time and ensure consistency. Customize the theme (colors, typography) a bit to avoid a stock look – even a personalized color scheme or logo can make it feel unique.
- **Responsive & Accessible:** Ensure the layout is responsive so it works on different screen sizes (you can use flexbox/grid and test resizing the browser). Also pay attention to accessibility: use proper HTML semantics, ARIA labels, and high-contrast colors. A recruiter who notices accessible design will know you sweat the details. Many UI libraries help with this out of the box.
- **UX Considerations:** Provide a smooth user experience. This means:
- Meaningful loading states (spinners or progress bars when data is loading or actions are in progress).
- Clear notifications or messages on success/failure of actions (e.g., "Device added successfully" toast, or "Failed to fetch data" alerts with potential reasons).
- Organize content with intuitive navigation – perhaps a sidebar or navbar with sections like "Devices", "Automation Tasks", "Monitoring", "Settings", etc.
- If an action might take time (like running a network command), consider implementing async feedback. For instance, you could show a job in progress and auto-refresh when done, or use web sockets to get live updates (more advanced).
- **Visualizations:** Include at least one compelling visual element, such as a chart or graph. For example, a line chart of network latency over time, or a pie chart of device status (up/down), or even a network topology graph (if you're adventurous, D3.js or Vis.js could create a network node-link diagram). Visuals make your project memorable and show you can handle front-end data rendering.
- **Polish:** Finally, polish the little things: custom favicon and title for the app, a nice landing page or login screen, tooltips on icons, etc. These subtle touches, while not difficult, create a perception that the project is "complete" and thoughtful. Many candidates might have basic projects, but few will take the time to refine the UX – doing so puts you a cut above. Consider also adding **user documentation** accessible from the UI (like a "Help" modal or linking to your project wiki) – this loops back to knowledge sharing.

By creating a frontend that is both attractive and user-friendly, you demonstrate an understanding that **presentation matters** in engineering projects (as one portfolio guide notes, a portfolio should be visually appealing and easy to navigate [12]). This level of UI/UX effort will make your project not just functional, but enjoyable to review.

Backend Design & Best Practices

The backend is the engine of your project, and it needs equal attention for robustness and clarity:

- **Framework & Structure:** If using Python, a framework like **Flask** or **FastAPI** is great for a lean REST API, while **Django** provides more out-of-the-box (ORM, admin panel) if needed. If using Node, **Express** is straightforward, whereas **NestJS** might give more structure (it's similar to Angular's style, with controllers, services, etc.). Whatever you choose, structure the project for **maintainability**:
- Organize code by feature/domain (e.g., `devices` module, `monitoring` module, etc.), each with its controllers/routes, services (business logic), and models (database schemas).
- Apply design patterns where appropriate. For instance, use a service layer to handle complex logic rather than writing everything in route handlers. This separation makes it easier to test and evolve.
- If the project grows, consider a plugin or modular architecture. Perhaps the network drivers (that talk to different device types) could be loaded as separate modules or classes – demonstrating you can design for extension.
- **API Design:** Adhere to RESTful conventions: use proper HTTP methods (GET for retrieve, POST for create, PUT/PATCH for update, DELETE for remove), and proper status codes. Design intuitive endpoints, e.g., `GET /api/devices` to list devices, `POST /api/devices` to add one, etc. Provide useful responses and error messages (and handle errors gracefully on the server side, returning JSON with error info rather than full stack traces).
- Consider versioning your API (e.g., prefix with `/api/v1/`) as a sign of forward-thinking design.
- If you're comfortable, you might implement GraphQL instead/as well – but REST is usually sufficient and more familiar to many.
- **Network Automation Logic:** This is a special part of the backend. Depending on your approach:
- If using actual libraries (Netmiko, etc.), make sure to handle exceptions (network timeouts, auth failures) so that a failure doesn't crash the app. Wrap calls in try/except and return clear error states that the frontend can display.
- If long-running, run these tasks asynchronously. For example, integrate a task queue like **Celery** (with Redis or RabbitMQ broker) for Python to handle jobs in the background. Or use Python's `asyncio` with async frameworks (if using FastAPI, you can call async functions to do network calls concurrently). In Node, you might use background workers or simply offload heavy tasks to a separate process/service.
- Ensure that multiple automation tasks can run without blocking each other – this is important for demonstrating an SRE mindset (concurrency and efficiency).
- Logging: log key events in the automation process (e.g., "Deployed config to Router A") either to console or a file, with timestamps. This will be useful for debugging and for the monitoring aspect.
- **Database & Models:** Design your schema thoughtfully. For instance, a `Device` table for network devices, maybe a `Task` or `Job` table to track automation jobs history, and a `Metric` table if storing time-series data yourself (though you might rely on Prometheus for that). Define relationships (maybe devices belong to a Site, etc., if you want to show relational modeling). Use migrations (Alembic for SQLAlchemy, Django migrations, etc.) to manage schema changes – this is another sign of professionalism.
- **Testing:** Back-end code especially should have **automated tests**. Create unit tests for your utility functions (e.g., a function that parses device output) and possibly integration tests for API endpoints (spinning up an in-memory version of the server and hitting it). Having tests not only improves quality but also impresses recruiters; as you noted, many projects lack them, so this is low-hanging fruit to stand out. The referenced DevOps project explicitly included *"working unit and integration*

tests, which are not always available in other hello-world type apps" [6] . Aim for a decent coverage (you don't need 100%, but cover core logic). Use CI to run these tests on each commit (discussed later).

- **Clean Code Practices:** Follow PEP8 style in Python or ESLint/Prettier in JS. Variable and function names should be clear. Break down large functions into smaller ones. Essentially, anyone reading your backend code should find it professional and maintainable. It might help to have a friend or colleague do a code review for you, to get feedback on code clarity – treat it like a real code review process.
- **Security Practices:** Even if it's a demo app, show awareness of security:
- Don't store secrets (passwords, API keys) in code or repo – use environment variables or a config file that's .gitignored. Document how to configure these. Perhaps integrate a tool like **dotenv** to load configs, or support a configuration file for advanced users.
- Validate inputs on the server side (never trust just client-side validation). For example, if an API expects an IP address, you can include a check to ensure it's a valid IP format, etc.
- If you implement auth, use proper password hashing (bcrypt/scrypt) and JWT or session cookies securely.
- Consider authorization if relevant (maybe certain users can only view but not perform actions – this might be overkill, but even a mention that you've thought of roles will show depth).
- Protect against common vulnerabilities: e.g., if using SQL, use parameterized queries or ORM to avoid injection; if your app allows any user input to reach a shell command (maybe calling a ping), be careful to sanitize inputs.
- These might not all be visible to a casual user, but you can mention them in documentation ("Security: implemented X to ensure Y") to signal that you are thorough.

In summary, the backend should reflect production-grade thinking: well-structured, tested, secure, and scalable. You want to show that you can build the **engine** of an application that is reliable and maintainable. This is your chance to display the "software craftsmanship" side of competency, which nicely complements the learning and knowledge-sharing aspects.

DevOps, CI/CD & Infrastructure

To truly impress for DevOps/SRE roles, **how you build and deploy the project** is as important as the code. You should treat your project as if it will be run by others or even in a production environment:

- **Version Control & Repo Setup:** Use Git from day one. Structure your repository (or repositories) clearly – for example, one for backend, one for frontend, or a monorepo with folders `/frontend` and `/backend`. Make regular commits with clear messages. It's great to showcase a clean commit history; it shows your working style. Also consider using branching (e.g., feature branches, a `main` or `master` branch that's always stable). You could adopt a Git branching strategy (like GitFlow or GitHub flow) to simulate team practices. It might be overkill to enforce for a solo project, but even mentioning it in docs ("Using GitHub flow for branching and pull requests to myself for code reviews") can stand out.
- **Continuous Integration (CI):** Set up an automated build and test pipeline. GitHub Actions is a convenient choice (free for public repos). Configure it to run on each push:
- For backend: run the test suite, maybe run a linter (flake8/black for Python, ESLint for JS) to enforce code quality.
- For frontend: you can also run build/test (e.g., if using React, run `npm test` for any front-end tests, and ensure the production build compiles with `npm run build`).

- If all tests pass, have it report success. If not, fix issues. This CI status can even be shown as a badge on your repo's README. Having a green "CI build passing" badge immediately signals quality.
- **Continuous Delivery/Deployment (CD):** If possible, automate deployment as well. For example, you could use Actions to build Docker images and push to a registry, or deploy to a cloud service. If you plan to host a live demo of your project (which would be fantastic), consider using a service like **Heroku, Netlify, Vercel, or Render** for free-tier hosting. For an SRE-oriented approach, you might set up a pipeline that deploys to a personal server or cloud instance using tools like **Terraform** or Ansible. Even if it's just pushing to an AWS EC2 or DigitalOcean droplet, automating that (maybe via SSH in CI or using cloud provider APIs) demonstrates real DevOps skill.
- At minimum, provide **deployment scripts/instructions**. For instance, a Docker Compose file that can spin up the whole stack (database, backend, frontend). This makes it easier for recruiters to try it out locally. In fact, from the example project: *they provided a docker-compose setup so one could launch the demo in minutes*. You can do similarly – a one-command startup is gold for impressing (because many side projects are a pain to run).
- **Containerization:** Use **Docker** to containerize the application. You can have separate Dockerfiles for frontend and backend. This not only shows you know how to containerize apps, but also helps with deployment (you can ensure it runs anywhere). Write a `docker-compose.yml` to run multi-container (DB, backend, frontend all together). Ensure to use environment variables in Docker for configs (and document them). You can also demonstrate multi-stage builds in Docker for efficiency (e.g., build the React app then serve with Nginx, etc.). This container work directly ties to DevOps skills.
- **Infrastructure as Code:** If you plan to deploy to cloud, use Terraform scripts to define infrastructure (networking, VMs, containers, etc.). For example, a Terraform config for an AWS ECS or EKS cluster to run your containers, or for DigitalOcean App Platform. If not actual cloud, you could use a local Kubernetes cluster (maybe include a Helm chart for your app). These are advanced steps, so prioritize based on your comfort – but even a basic AWS deployment via Terraform that sets up an EC2 with Docker can be a big differentiator.
- **Observability Implementation:** Set up monitoring and logging for your deployed app:
- **Logging:** Use a centralized log if possible. For instance, if using Docker, maybe have logs go to stdout and aggregate with a tool (or just instruct users to check logs). Or integrate with a service like Logz.io or ELK stack if you're ambitious (could be too much, but an Elasticsearch+Kibana for logs would wow some). At least, make sure your app logs have consistent format (timestamp, severity, etc.) – you can use a logging framework.
- **Monitoring:** As discussed, incorporate Prometheus for metrics collection. You could run a Prometheus container via docker-compose as well, scraping your app. Then run Grafana to visualize. If full Prom/Grafana integration is heavy, you can simulate by logging metrics or exposing an endpoint and using your UI to show metrics. The key is to show you know how to measure app performance. **Set up alerts** if possible – even a simple one like "if app response time > X, print an alert". In docs, mention what you'd monitor (CPU, memory, errors, etc.) and maybe include sample Grafana screenshots (images in your README) to show it working.
- **Reliability & Scaling:** Address how the app would scale or stay reliable:
- Possibly use a load testing tool (like JMeter, k6, or just a Python script) to simulate load and mention how it performed or what was optimized. For example, if you discover the app can handle N requests per second, note that, and maybe you added caching for improvement. Such insights show an SRE mindset.

- Implement redundancy if applicable – e.g., the backend could be stateless (session in DB, etc.) so it can run multiple instances behind a load balancer. If using Kubernetes, you could demonstrate horizontal scaling (replicas of your backend).
- Error handling: Make sure any background jobs that fail don't crash the whole system. Possibly use a message queue that can retry jobs, or at least catch exceptions and return error statuses gracefully.
- **DevSecOps:** Security in the pipeline – you could integrate some static analysis or dependency vulnerability scan in CI (GitHub has Dependabot and code scanning, or use a tool like Bandit for Python security lint, or npm audit for Node). You could also scan your Docker images for vulnerabilities (Anchore or Trivy). These might be overkill, but mentioning them shows you're aware of secure development practices. For instance, the DevOps roadmap from Reddit suggested adding SAST, DAST, image scanning in the pipeline for advanced learners [13] [14]. Even if you only implement one (say, `npm audit` or `pipenv check`), it's worth it.

In essence, this part of the project transforms it from just code into a **living system** that could run in the real world. It demonstrates the "operations" side of DevOps/SRE. Many candidates might have a good app, but few will containerize it, set up CI/CD, and deploy it with monitoring. Doing so will unequivocally make you **"a cut above the rest"** in the eyes of technical recruiters, because it proves you can not only build software, but also deliver and maintain it in a production-like environment.

Documentation & Knowledge Sharing

Given the emphasis on "sharing knowledge above all," documentation is critical – both for the project's users and to showcase your communication skills:

- **Comprehensive README:** Your repository's README.md should be a mini-website in itself. Start with a clear description of the project (what it does and why it's useful). Include a screenshot or two of the UI to catch attention. Provide a **table of contents** for easy navigation (installation, usage, architecture, etc.). Key sections to include:
- **Features:** List the major features of the project, especially those that align with network automation and SRE skills (e.g., "Automates network config backups," "Real-time monitoring dashboard," "CI/CD pipeline integrated"). This gives a quick overview of what's impressive.
- **Installation/Setup:** Step-by-step instructions to run the project. If using Docker, this might be as simple as "install Docker and run `docker-compose up`." Also mention any prerequisites (accounts, API keys, etc., if any). The easier you make it to run, the more likely someone (recruiter or dev) will actually try it. Remember the example: they allowed running without heavy DB setup using SQLite for convenience [8] – you can do similarly or provide a ready-made demo environment.
- **Usage Guide:** Explain how to use the application. Perhaps walk through an example: "To add a device, go to this page, fill the form, then you can run X action..." including screenshots or terminal output. This doubles as showing that you yourself understand the user's perspective.
- **Architecture:** This is where you shine a light on your design thinking. Describe the system architecture – possibly with a diagram (you can create a simple diagram showing frontend, backend, DB, external services). Outline the tech stack (languages, frameworks, libraries) and *why* they were chosen. For example, "Backend uses FastAPI (Python) because of its async support and great documentation, allowing fast network I/O for device communication [1]. Frontend uses React for a dynamic UI and widespread industry use. PostgreSQL chosen for reliable relational storage of device data," etc. Also discuss how components communicate (REST API, websockets for live updates, etc.).

This explanation of why and how is a major knowledge-sharing aspect – it educates the reader on how to approach building such a system.

- **Challenges & Lessons:** Include a section (or weave into a blog post) about challenges you encountered and how you solved them. Perhaps you struggled to get a certain library to work with an IPv6 address – mention how you resolved it. Or you optimized API response by doing X. Reflect on what you learned (e.g., "Implemented a circuit breaker for network calls to handle flaky connections – this taught me about graceful degradation"). This kind of storytelling (remember: *"describe challenges faced and solutions implemented"* [10]) humanizes the project and highlights your problem-solving skills.
- **Future Improvements:** Show that you have a forward-looking mindset by listing some features or improvements you'd like to add. This might include things from the earlier wish list (like "Add support for vendor X devices" or "Implement chaos testing as next step"). It tells reviewers that you know the project isn't perfect and you have the initiative to continue refining it.
- **Contributing Guide:** Given the open-source best practice focus, if you open-source the project, include a CONTRIBUTING.md. This can outline how others can file issues, suggest improvements, or contribute code (even if you don't expect contributions, having it demonstrates you know how to run an open project [15] [16]). Similarly, a CODE_OF_CONDUCT.md is a nice touch for inclusivity (templates are available for this).
- **License:** Choose an open-source license (MIT or Apache 2.0 are common). It's a small thing, but important for any public code – and it shows professionalism.
- **Credits/Acknowledgments:** If you referenced any tutorials, used open-source libraries, or took inspiration from other projects, give them credit. This shows humility and that you engaged with the community in building knowledge.
- **Wiki or Docs Site:** For very extensive documentation, you might use a GitHub Wiki or a docs site generator (like MkDocs or Docusaurus) to host documentation (for instance, NetBox has a `docs/` folder and uses ReadTheDocs [17]). This could hold more detailed technical docs: e.g., API documentation (if you didn't embed Swagger/OAS), design docs for complex portions, or user manuals. While not every recruiter will read these, just having them available signals thoroughness.
- **Inline Documentation:** Ensure the code itself is well-documented. Use docstrings in functions/classes to explain their purpose, especially for any tricky part of the network logic. Clearly comment sections that might be non-obvious. This not only helps you later but also anyone looking at the code. Adopt a consistent style for this (Google style or reStructuredText for Python docstrings, JSDoc for JS, etc.). If someone were to generate documentation from your code, it should be straightforward.
- **Knowledge Sharing Externally:** Beyond the project repo, you can amplify knowledge sharing by writing about the project in a blog or LinkedIn post. For example, a Medium article titled "How I Built a Network Automation Dashboard with React and Python" could summarize the project and key learnings. This not only proves you did the work, but you can also communicate it – a valuable skill. You might also consider doing a short video demo and uploading it (unlisted YouTube to share with recruiters). A live demo site is ideal, but a video is a good fallback that guarantees they see it in action without setup.
- **Education Focus:** Because "learning" is a north star, you could integrate a learning component in the project for others. For instance, provide sample data or a tutorial mode in the app. Or include a directory of example scripts (maybe a `/examples` folder with sample network scripts in Python) that users can try. Treat your project as if you're mentoring someone through it. This mindset will naturally make your documentation and project more inviting.

When your documentation is strong, you **stand out immensely**. Many developers neglect this, so a recruiter seeing a detailed README and wiki will immediately know you're someone who goes above and beyond. It reflects passion and professionalism. As one resource emphasized, the portfolio transforms from a mere resume extension into a **"captivating showcase of your talent"** when you invest time in it [18]. Documentation is the narrative that ties your skills together – don't skimp on it.

Implementation Roadmap

To execute this ambitious project without getting overwhelmed, break it into phases with clear milestones. Below is a proposed rollout plan:

Phase 1: Planning & Design
- **Define Scope:** Write down the core features you want in the first version (MVP) – for example, "Manage devices (CRUD), run a sample automation (like ping), view a health dashboard." Also list extended/optional features separately (so you have a prioritized backlog). This keeps you focused.
- **Technical Research:** Decide on primary tech stack components (language, frameworks, DB, any specific libraries for network tasks). Research feasibility of any unknowns (e.g., how to integrate Prometheus, or how to simulate a router if you need to). Create a simple diagram of the architecture.
- **Project Setup:** Initialize version control (Git repo). Write a brief README with the project idea and roadmap to give future visitors context (you will expand it later, but even a stub is good).
- **Design Review (PDR):** Before coding, do a mini "preliminary design review" – this could just be with yourself or a peer. Go through the planned architecture and tech choices to see if anything might be a roadblock. Adjust plans accordingly. Essentially, catch flaws on paper first.
(Deliverable: Project plan in repo, initial commit with README and perhaps empty framework files.)

Phase 2: Backend Foundation
- **Scaffold the Backend:** Set up the project structure for your backend. For instance, create a Flask app with a couple of blueprint modules (even if they're mostly empty). Or generate a new Django project. Ensure it can run (even if it just returns a "Hello World" on an index route).
- **Database Setup:** Set up the database connection early. Create the data models for the most crucial entity (say, Device). Apply migrations if using an ORM. Test that you can connect to the DB and perform a simple query.
- **One Core API:** Implement one simple API endpoint end-to-end, for example "GET /api/devices" that returns an empty list or a static sample. This touches all layers (routing, possibly DB, serialization) and ensures the stack is sound. Write a basic unit test for this endpoint (or the underlying function).
- **Set up CI Pipeline (initial):** Configure GitHub Actions (or another CI) to run a test workflow. Even if you have just one test, getting the pipeline in place now is wise. Also add linters (e.g., run flake8/pylint) to enforce code style from the start. This prevents bad habits and shows that quality is built-in from the beginning.
(Deliverable: Basic backend running locally, CI passing on a trivial test, code structured.)

Phase 3: Frontend Foundation
- **Scaffold the Frontend:** Create the React app (using create-react-app or Vite or Next.js, etc.). Set up the project structure (components directory, services/api directory for calling backend, etc.). Integrate a UI library (install Material-UI or others) and verify you can use a component.
- **Basic UI & Routing:** Implement a simple homepage or dashboard component. Set up a router (using React Router, for example) with a couple of routes corresponding to future sections (Devices, Dashboard,

etc.). They can be empty pages with “Coming soon” text for now. This is to establish navigation early.
- **API Call Test:** From the frontend, try calling the backend API you made in Phase 2. For instance, have the Devices page fetch `/api/devices` and display the result (even if empty). Handle the fetch with proper error handling. This will surface any CORS issues (configure CORS on backend as needed). Once this works, you have the full vertical slice: frontend calling backend using real HTTP.
- **Frontend DevOps:** Optionally, add the frontend build/test to your CI pipeline as well (e.g., run `npm run build` to ensure no compile errors). Also consider setting up a development Docker environment now: perhaps a Dockerfile for the backend and one for frontend (the frontend Dockerfile can simply run an nginx serving the build). This may be refined later, but starting now can catch issues early.
(Deliverable: Basic React app with routing and a sample API integration, CI updated to include frontend build.)

Phase 4: Core Features Implementation
Now that the skeleton is in place, build the main features one by one: - **Device Management:** Implement the backend endpoints for device CRUD (Create, Read, Update, Delete devices) and the frontend UI to list devices and add a new device. Focus on getting a simple form working and data saving to the database. This will flex both your form handling on the frontend and database handling on the backend. Don’t worry about automation yet; treat this as standard CRUD functionality. Add form validation (e.g., IP address format check). After completing, you can show that you have a working mini-app (managing items in a list). Write tests for the backend logic (e.g., a test for adding a device with valid/invalid data).
- **Automation Action (e.g., Ping or Fetch Config):** Choose one representative network automation task and implement it end-to-end. For example, a “Ping Device” action: - Backend: a route like `POST /api/devices/{id}/ping` which triggers a ping. For now, it could actually call the system ping command or use a library, or just simulate by waiting 1 second and returning “success”. The key is to structure it as if it were doing the real thing (perhaps have a NetworkService class with a `ping_device(ip)` method). If it’s a quick action, you can do it inline; if it might be long, implement a background job (this could be a good point to introduce a job queue and demonstrate async handling). Store the result in the DB or return directly. - Frontend: Add a button on the device list or detail page: “Ping”. When clicked, call the API and show the result (e.g., “Host reachable, 20 ms latency” or an error message). Provide user feedback while pinging (spinner or disabled button). Once this is working, you’ve demonstrated actual network interaction. - Expand on this with another action if time permits, like “Fetch Config” (which could return a dummy configuration text), to show multiple capabilities. - **Monitoring Dashboard:** Implement the basics of the monitoring page. For instance, have the backend provide an endpoint `/api/metrics` that returns some stats. You can accumulate these stats in memory for now (like count of pings done, average ping time, etc.), or integrate with a real metric source. On the frontend, use a chart library to display one of these metrics over time. This could be as simple as a line chart that adds a point each time you fetch the metrics endpoint (simulate real-time update). If you set up Prometheus, you might instead query Prometheus for data. But even a manually updated chart from an API will impress if it looks real-time.
- **Iterative Testing:** After each feature, ensure tests are written or updated. Run your entire test suite frequently. Also test the app manually – click around the UI, try edge cases (like invalid form input, or ping a non-existent IP to see error). Fix any bugs. - **Commit and Document:** At the end of this phase, you essentially have an MVP. Update the README usage section with instructions on these features. You might even release a “v1.0” of your project on GitHub to mark this milestone (recruiters will see the Releases and appreciate a versioned approach). *(Deliverable: MVP completed – devices can be managed, one or two automation actions work, a basic monitoring view exists. Tests covering core logic, documentation updated for usage.)*

Phase 5: Advanced Features & Hardening

With the core in place, add the extra mile features and harden the system: - **Authentication & Security (if applicable):** If you decided to include user accounts, implement it now. Use a secure auth system (perhaps JWT tokens for a single-page app: user logs in via `/api/login`, gets token, frontend stores it and sends in headers). Ensure password hashing and proper auth checks on protected endpoints. If skipping auth, consider at least restricting dangerous actions by a simple token or config (so not *everyone* can run it if hosted). - **Role-based Access (optional):** For example, an "admin" vs "read-only" user just to demonstrate knowledge of permission handling. This could be overkill, so do only if time permits and relevant. - **Error Handling & Validation:** Do a sweep of your backend to improve error messages and input validation. Make sure every API endpoint returns appropriate errors for bad input and doesn't just crash. For instance, handle device not found, or ping timeout scenarios clearly. On the frontend, ensure these error messages are displayed to the user neatly. This level of robustness is something many skip – doing it shows reliability focus. - **Optimize and Refactor:** Review your code for any inefficiencies. Perhaps caching the device list in memory to avoid DB hits on every request (if that's a concern), or using pagination if the list grows. Maybe refactor some functions for clarity or to reduce duplication. Ensure your logger is used consistently everywhere important. - **Extend Monitoring:** If you started simple, maybe now integrate a real monitoring tool. For example, run a Prometheus server (maybe as a Docker container included in docker-compose) and configure your app to expose `/metrics`. Then polish your dashboard to fetch data from Prometheus (could use its API) or include Grafana for visualizations. Alternatively, implement a simple custom metrics store: e.g., log the last N ping results and show stats. The aim is to deepen the SRE aspect now that basic functionality is done. - **Documentation Expansion:** As new features (auth, etc.) are added, update docs. Also, now is a good time to flesh out the Architecture and Lessons Learned sections of your README, because you have the context of building it. Consider writing a separate **design document** for the project (outlining trade-offs, how you ensured reliability, etc.) and include it in the repo. This could even be the PDR document updated with reality vs. plan. - **User Testing:** If possible, have someone else follow your README to setup and use the project. Their feedback will highlight if something is confusing in the docs or if there are setup issues on a clean system. Fix those issues to improve usability. *(Deliverable: Enhanced feature set (auth, etc. if chosen), robust error-handling, refined code, extended monitoring, and updated documentation reflecting the full system.)*

Phase 6: DevOps & Deployment

Now, ensure the project can run in different environments and show off the DevOps integration: - **Dockerization:** Finalize Dockerfiles for each component. Optimize them for production (small image sizes, using multi-stage builds). E.g., build the React app and serve static files with Nginx, or use Node to serve if simpler; use a lightweight base image for Python (alpine or slim). Test that `docker-compose up` successfully brings up the whole app. This is your easiest way to let others try it. - **Continuous Deployment:** If you set up CI, extend it to publish artifacts. For instance, use GitHub Actions to build and push Docker images to Docker Hub on each release or commit to main. Or auto-deploy to a host. This might involve secrets (like Docker Hub creds or cloud API keys) – manage them via GitHub Secrets. Test the pipeline by doing a trial release. - **Cloud Deployment:** Deploy the app to a cloud service or server. Options: - Simple: Launch a VM (AWS EC2, Azure VM, DigitalOcean droplet), install Docker and run the compose. This proves it works on a remote server. You can expose it so that there's a URL where the app is live. (Be mindful of security – at least put a simple auth if it's public). - Modern: Deploy to a container service (AWS ECS or Fargate, Google Cloud Run, etc.), or orchestrate with Kubernetes (maybe use a free K8s like K3s on a VM, or kind cluster for demo). For Kubernetes, create manifests or a Helm chart for your app. Demonstrating K8s knowledge is a big plus for SRE roles. - Document whatever you do in a *Deployments* section of README (e.g., "Deployed on AWS using Terraform – see infra/terraform folder for configuration" or "Hosted on

Heroku at [link]"). If you can provide a live link to the app (even if it's just up for a while during your job search), it massively impresses recruiters because they can click and see it working immediately. - **Infra as Code:** If you used Terraform or similar, include those config files in the repo (in an `infra/` directory). Mention how to use them (like `terraform apply` steps). This again underscores that you treat infrastructure as part of the codebase. - **Backup/Recovery:** One thoughtful touch: provide a way to backup data or handle updates. For example, instructions on how to dump the database, or migrate it. SREs think about disaster recovery – a brief note like "database dumps can be taken with this command, and the app can be restored by running migrations on a new instance" shows you considered maintenance. *(Deliverable: Docker images running the app, possibly a live deployment, CI/CD automating build/deploy, infrastructure code included.)*

Phase 7: Final Polish & Review
At this stage, the project should be fully functional and deployed. Do a final round of polishing: - **UI/UX Polish:** Refine CSS and visuals. Ensure consistency in typography, spacing, and element alignment. Maybe add a dark mode toggle (just for fun and attention to detail). Ensure mobile responsiveness if relevant. Little tweaks can elevate the look (e.g., smooth animations on state changes, a custom logo or icon set for the app). - **Performance Tuning:** Run some quick performance tests. Use browser dev tools to see if any API calls are slow or if bundle size is huge. Optimize where possible (enable gzip compression on your server, maybe code-split the frontend if it's large, etc.). On the backend, check for any obvious slow queries or memory issues. This shows you care about efficiency. - **Accessibility & QA:** Do an accessibility audit (browser dev tools can simulate this) – fix any serious issues (like missing alt tags, etc.). Also do a cross-browser check (Chrome, Firefox, maybe Safari) to catch any frontend quirks. - **Final Documentation Touches:** Make sure all sections of the README are up-to-date. Add the badges (build status, license, etc.) at the top for quick info. Possibly add a section "Why this project is awesome" where you bullet-point the unique selling points (this can be a summary for recruiters who skim: e.g., " Full-stack app with React/Flask and Postgres", " Infrastructure as Code with Terraform + CI/CD pipeline", " Comprehensive docs and unit tests (80% coverage)", " Demonstrates network automation via simulated device actions", etc.). This TL;DR can be very effective. - **Create a Demo Video:** Record a short screencast (using a tool like OBS or even Zoom recording) of you using the app – click through features and narrate or caption what's happening. Keep it 2-3 minutes. You can share this via a link in the README ("Watch Demo"). This often bypasses the hassle of running the app for busy reviewers and lets you control the narrative of what's shown. - **Peer Review:** If you have a mentor or friend in the industry, ask them to review the project (code or just the repo). They might point out improvements or questions a recruiter might ask. Use that feedback to refine final details. - **Finalize Version & Release:** Tag a version (v1.0) and briefly describe it in GitHub Releases. This snapshot is what you'd essentially send out. You can continue to iterate (v1.1, etc.) if you have new ideas or to fix bugs as they come up. *(Deliverable: Project is in a showcase-ready state – polished UI, all features working reliably, documentation complete, and extras like demo media available.)*

Following this roadmap ensures a structured approach where at each stage you have a working product (even if minimal), and you progressively add complexity. Importantly, you also integrate best practices *as you go* (CI, tests, docs, etc.), rather than as an afterthought, which is the hallmark of a good engineer. Each phase is an opportunity to commit and push code, which means your repository will show a timeline of progress – another great signal of consistent effort.

Conclusion

By adhering to this plan, you will build a project that is not only technically impressive but also thoughtfully presented. It's a chance to **demonstrate competency across the board** – frontend finesse, backend solidity, automation savvy, DevOps rigor, and communication clarity. You'll be learning at every step (trying new stacks, tools, and techniques) and immediately applying that knowledge, which aligns perfectly with the "north star" of continuous learning and knowledge sharing.

When complete, this portfolio project will serve as a **personal case study** of your abilities: any recruiter or hiring manager reviewing it will see clean code, a useful application, and a developer who cares about doing things the right way. You'll have gone above and beyond the average, by not just building an app that works, but one that could plausibly be used in a real-world scenario with confidence – something companies highly value.

Remember to keep reflecting on what you're learning as you build (perhaps maintain a short journal to later convert into a blog post about the experience). In interviews, you'll then have rich stories to tell about decisions you made, challenges you overcame, and how you kept the "north star" in sight throughout. This combination of a stellar project and the narrative behind it will strongly reinforce that you are *exactly* the kind of passionate, competent engineer that top companies want to hire.

Good luck, and enjoy the journey of building something truly standout!

Sources:

- An open-source DevOps learning project emphasized using a modern multi-language tech stack, iterative development, built-in tests, a service dashboard, and thorough documentation to create a realistic learning app [1] [19] [7]. These practices inspired many of the recommendations above.
- Recruiter insights highlight the importance of clean code, solid architecture, and tailoring projects to target roles. Portfolios should showcase problem-solving and relevant skills, not just list technologies [20] [5]. We applied this by aligning features with Network Automation and SRE skill sets.
- The design and polish targets were informed by examining successful tools like NetBox, which provides a robust UI and API as a network source-of-truth (a gold standard in network automation) [3], as well as by following general portfolio best practices around storytelling and presentation [10] [12].
- Advanced DevOps/SRE concepts such as observability, defining SLI/SLOs, and chaos engineering were suggested to truly differentiate the project, echoing the kind of "above and beyond" work that expert practitioners recommend for mastering these fields [21] [4].

---

[1] [4] [6] [7] [8] [13] [14] [19] [21] Open Source Devops Learning App with 15 Projects to build in 2025 : r/devops
https://www.reddit.com/r/devops/comments/1hvpejm/open_source_devops_learning_app_with_15_projects/

[2] [5] [10] [11] [12] [18] [20] What Recruiters Look for in Developer Portfolios
https://pesto.tech/resources/what-recruiters-look-for-in-developer-portfolios

3 9 15 16 17 GitHub - netbox-community/netbox: The premier source of truth powering network automation. Open source under Apache 2. Try NetBox Cloud free: https://netboxlabs.com/products/free-netbox-cloud/

https://github.com/netbox-community/netbox