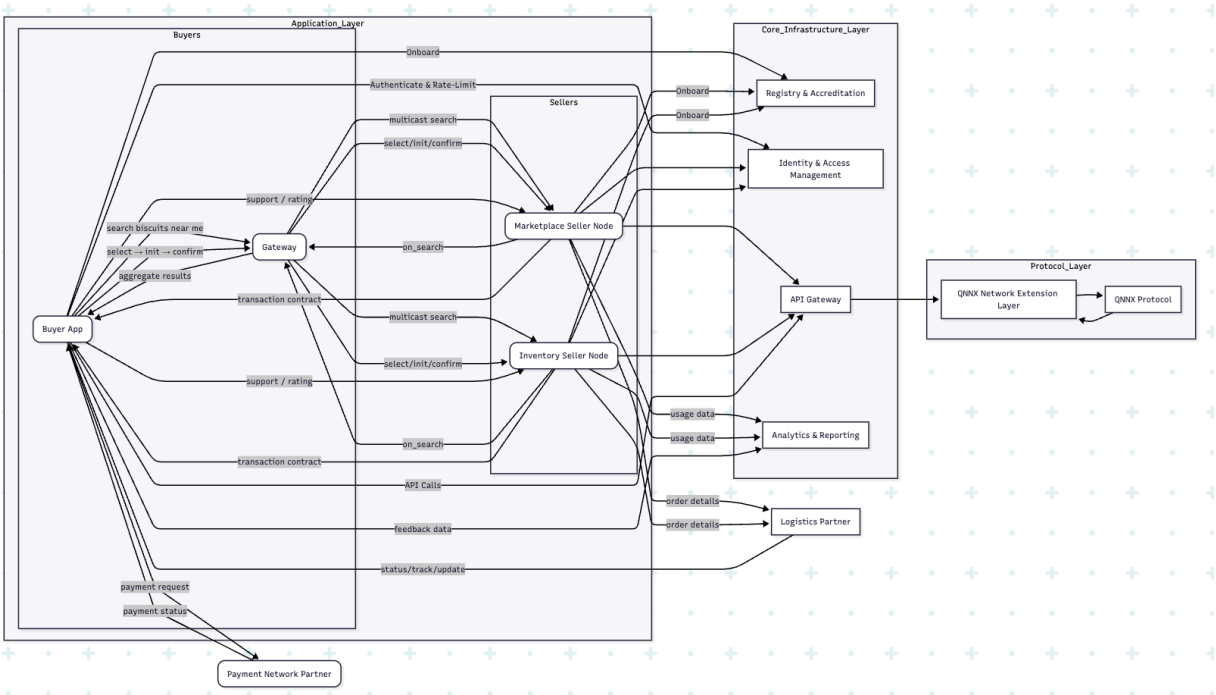
Report: Development and Deployment Strategy for QNNX's Multi-Layered System Architecture

Executive Summary

This report outlines a detailed development plan for implementing the **3-layer architecture** of the proposed system, focusing primarily on the **Core Infrastructure Layer**. It compares both **fully local** and **AWS-based** deployment models, including a breakdown of tools & technologies, required engineering roles, development responsibilities, time and cost estimates, and recommendations for optimal implementation strategy.



System Architecture Overview

Layered Breakdown

1. Core Infrastructure Layer
    - - Registry & Accreditation
    - - IAM
    - - API Gateway
    - - Analytics & Reporting
2. Protocol Layer
    - - QNNX Protocol
    - - Network Extension Layer
3. Application Layer
    - - Buyer App + Gateway
    - - Seller Nodes (MSN, ISN)
    - - Payment & Logistics Partners

(Note: In this report, most of the focus is designated to the core-infrastructure layer, the components within and attached to it)

TECHNOLOGY STACK (Component-Wise)

A. Core Infrastructure Layer

Component	Local Tools	AWS Cloud Services
Registry & Accreditation	PostgreSQL / SQLite, OpenSSL/CFSSL	DynamoDB + AWS ACM
IAM	Keycloak (Docker), JWTs	Amazon Cognito (User Pools + IAM)
API Gateway	NGINX or Kong (Docker)	AWS API Gateway (REST/HTTP)
Analytics & Reporting	Prometheus + Grafana, CSV logs	CloudWatch + S3 + Athena (optional)

B. Protocol Layer

Component	Tools/Technologies
QNNX Protocol	Python/Node.js/Rust
Network Extension Layer	Microservices (FastAPI, Express.js)

(Note: Following are some additional details regarding the involvement of “Network Extension Layer” within the parent “Protocol Layer”:

What Is the Network Extension Layer?

In simple terms, the **Network Extension Layer** is like a **smart post office** sitting between your system's core protocol and the apps (buyers, sellers, logistics, etc.). It helps **route messages**, **enforce logic**, and **keep services connected and talking smoothly**, without needing them to know all the details about each other.

What Does It Do (Simplified)?

- **Message Router**: It decides *where* each message should go — e.g., from the Buyer App to the right Seller Node.
- **Middleman Logic**: It adds smart behavior like checking "Is this seller online?" or "Does this seller sell the item?"
- **Format Translator**: It makes sure messages are properly structured before forwarding them — kind of like ensuring a package has the right address label.
- **Lightweight Control Center**: It can slow down or reject requests if they're too many or look suspicious.
- **Async Communication**: It can queue messages if a seller is busy — instead of requiring everything to happen instantly.

How Can You Build It (Small Scale)?

For a basic setup, you don't need complex infrastructure. You can build this layer with:

Tool	Purpose
FastAPI	To create small APIs to forward messages
Python dicts or **JSON configs**	For routing rules (e.g., match buyer to seller)
Simple Pub/Sub (e.g., Redis)	Optional: If you want to handle messages asynchronously
JWT Tokens	To validate if requests are allowed

(Note: No need for Kafka, Consul, or fancy load balancers at this stage. You just need smart functions that look at a request, apply rules, and send it to the right service.)

Example Flow

Let's say a **Buyer App** sends a "Search for Product X" request:

1. The **Buyer Gateway** receives the request and forwards it to your Network Extension API.
2. The extension service checks:
    - Which sellers are active?
    - Which ones offer Product X?
3. It forwards the request to those seller endpoints (say, 2 of them).
4. It collects the responses and returns the results to the buyer.

All this happens without the buyer or seller knowing the network details of the other — the Extension Layer handles it transparently.

Why It's Useful (Even for Small Teams)

- You **avoid hardcoding** logic in each app (Buyer or Seller).
- It's easier to **simulate real-world conditions** (like sellers going offline).

- You can **add features gradually** (e.g., load balancing, retries, filtering).

)

C. Application Layer

Component	Local Tools	AWS Tools
Buyer/Seller Apps	FastAPI / Flask / Express.js	AWS Lambda (Zappa/Chalice) / EC2
Buyer ➔ Seller Gateway	Express.js or Flask router	Lambda or containerized EC2
Logistics/Payments	Mock APIs	External APIs / AWS Lambda mocks

Minimum Team Roles & Responsibilities

Fully Local Setup (Min. 2 Engineers)

Role	Responsibilities
Backend Engineer	Build Buyer/Seller APIs, implement protocol logic, manage onboarding transactions
DevOps Engineer	Configure Docker/Docker Compose, NGINX, local monitoring tools

Optionally, security tasks (e.g. token gen, access rules) may be shared by DevOps.

AWS-Based Setup (Min. 2 Engineers)

Role	Responsibilities
Cloud Engineer	Provision AWS services (Cognito, Lambda, API GW, DynamoDB), set IAM roles and networking
Backend Developer	Write and deploy Lambda/container services, integrate with API Gateway and other components

Security engineering can be merged into the Cloud role at this stage.

Development Flow (Fully Local)

Stage	Engineer	Description
Environment Setup	DevOps	Install Docker, configure containers, local networks
Core Service Dev	Backend	Build protocol logic, Buyer/Seller APIs, messaging flows
IAM Config	DevOps	JWT/Keycloak setup
Monitoring Setup	DevOps	Prometheus + Grafana for metrics/logs
Simulation & Test	Both	Simulate flows: onboarding, messaging, purchase/order cycles

Development Flow (AWS-Based)

Stage	Engineer	Description
Infrastructure Setup	Cloud Engineer	Use AWS Console or Terraform/CDK to provision all required services
API Implementation	Backend	Create Lambda functions or deploy containerized APIs
IAM Setup	Cloud Engineer	Configure Cognito, define permissions, IAM policies
Gateway + Routing	Cloud Engineer	Set API Gateway paths, rate limiting, integrations
Observability Setup	Cloud Engineer	Use CloudWatch for metrics, S3 + Athena for deeper analysis (optional)
Integration & Testing	Both	Validate connectivity between services, simulate real-world flows

Time Estimates

Setup Type	Time Estimate	Notes
Fully Local	2–3 engineer-days	Fast setup, fewer external dependencies, manual coordination needed
AWS-Based	3–5 engineer-days	More robust infra, slight learning curve with IAM and deployment

Cost Expectations (Simulation Scale)

Type	Infra Cost	Engineer Time Cost

Fully Local	$0 (OSS stack)	~24–30 hours total
AWS-Based	$0 (Free Tier)	~32–40 hours total
Beyond Free Tier	~$1–$5/month	Low-scale simulation only

(Note: The time based estimates are governed by how experienced is the development team)

Local vs AWS Comparison

Factor	Fully Local	AWS-Based
Infra Cost	$0	$0–$5/month
IAM Complexity	Low (JWTs/Keycloak)	Medium (IAM roles, Cognito policies)
Observability	Prometheus + Logs	CloudWatch + Athena (optionally)
Iteration Speed	High (instant local testing)	Medium (Lambda deployment cycles)
Scalability	Low	High (horizontal scaling, managed infra)
Best For	Rapid prototyping, local dev	Simulated production, cloud-readiness

Strategic Recommendations

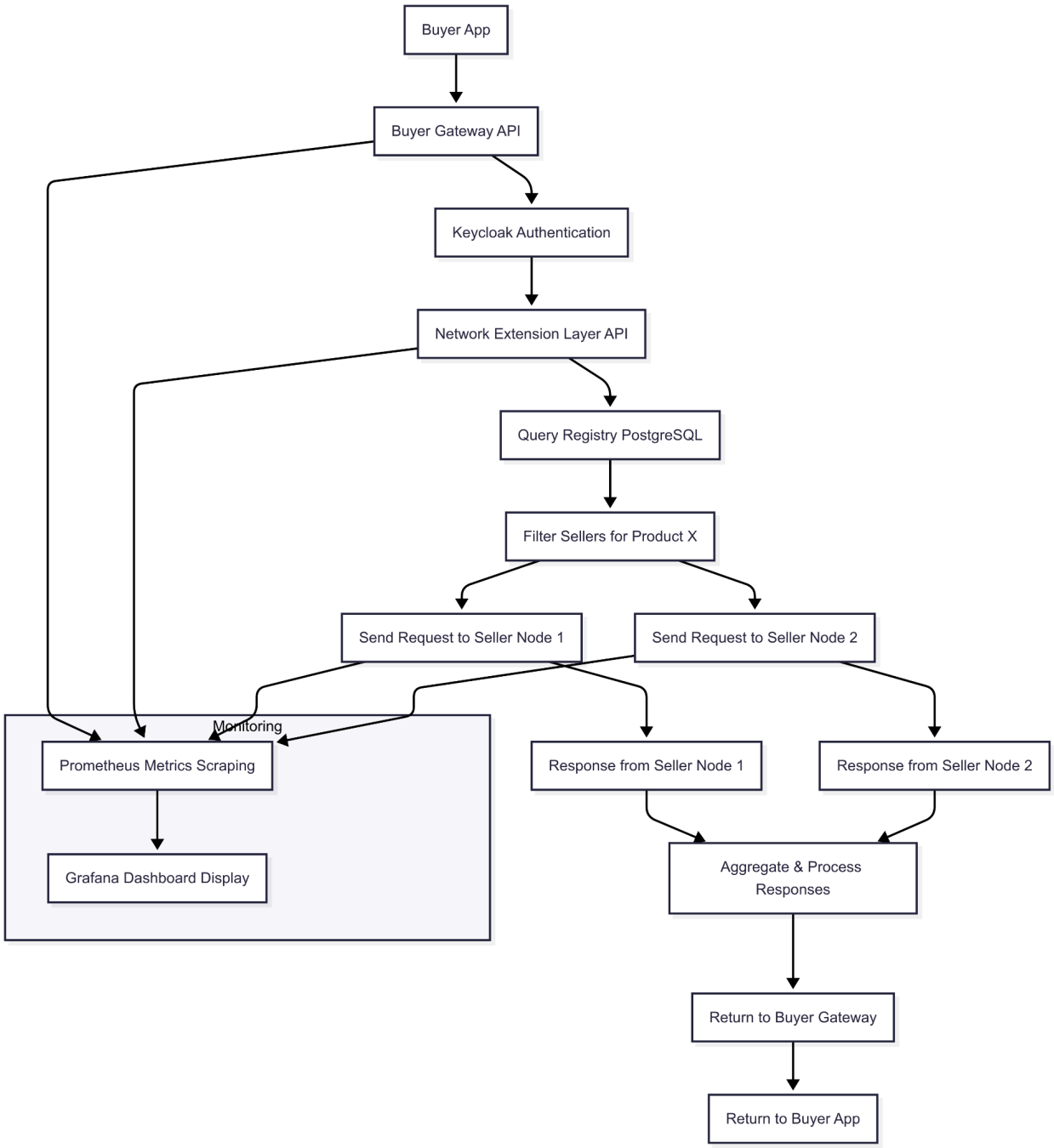
- **Start Local for POC**: Use Docker Compose + Keycloak + Prometheus/Grafana for rapid iteration.
- **Move to AWS gradually**: As IAM/auth and analytics become complex, port to Lambda, Cognito, API Gateway.
- **Tools to consider**:
    - **Zappa** or **AWS Chalice** for Python Lambdas
    - **Terraform/CDK** for infra automation
    - **OpenSSL** for local cert management

Validated Benchmarks & Assumptions

- **Team Size**: 2-person teams can simulate this stack realistically for demos/Pilot phases
- **Time Estimates**: In line with AWS starter guides and real-world engineering workflows
- **Tools Used**: Based on best practices and verified AWS/Open Source capabilities

Example Transactional Workflows

Fully-local



AWS Solution