Context-Aware Infrastructure Intelligence

How Multi-Source Synthesis Transforms AKS Performance Diagnostics

A CloudGo.ai Case Study: Azure Kubernetes Service Optimization with ReadyPay

1. Introduction

Infrastructure performance problems are rarely solved by the same tools that detect them. Monitoring dashboards surface symptoms (latency spikes, error rate increases, capacity warnings) but the root causes that connect those symptoms to actionable remediation live in architecture decision records, incident postmortems, compliance policies, and cost management systems that monitoring tools never access. The common result is for engineering teams to cycle through generic troubleshooting playbooks while the organizational context that would identify the actual fix sits in documents they never think to consult during an outage.

This case study demonstrates a fundamental thesis: infrastructure optimization recommendations change dramatically when live telemetry, resource configuration, compliance constraints, and operational history are synthesized alongside cloud best practices. We examine an AKS performance optimization scenario where full-context analysis identified three converging root causes that no single data source could reveal, prevented a scaling strategy that would have worsened the problem, and delivered an operationally precise remediation plan that a prompt-only analysis could only hypothesize about.

The subject company, ReadyPay, is a Series B fintech processing real-time payment authorizations on Azure Kubernetes Service. Their transaction engine experienced p99 latency spikes from 120ms to over 800ms during predictable peak windows, triggering SLA breaches with enterprise merchants. Leadership was divided: the CTO advocated horizontal scaling (add nodes, enable HPA and Cluster Autoscaler), while the infrastructure lead preferred vertical scaling (upgrade to larger VM SKUs). CloudGo.ai's context-aware analysis revealed that both positions were premature; the latency spikes were driven by three independent bottlenecks that would persist or worsen under either scaling strategy.

1.1 ReadyPay's Core Issue

Kubernetes infrastructure optimization operates at the intersection of compute scheduling, network architecture, IP address management, and organizational process. Each domain has its own monitoring tools, documentation practices, and operational expertise. CPU throttling metrics live in Container Insights. SNAT port allocation is tracked by Azure Load Balancer diagnostics. Subnet IP utilization is a property of the VNet configuration. PCI compliance requirements for network changes are documented in internal security policies. When a p99 latency spike occurs, the troubleshooting effort typically focuses on whichever domain the on-call engineer is most familiar with.

The result: technically reasonable recommendations that address one dimension while missing constraints from others. A recommendation to "enable Cluster Autoscaler and scale horizontally" is incomplete if it does not account for the subnet's remaining IP capacity, the SNAT allocation model's behavior when backend instances increase, or the PCI compliance review required before network topology changes can be implemented.

1.2 Success Criteria

Successful infrastructure optimization in this context required:

- *Root cause identification:* Moving beyond symptom-level observations to identify the specific mechanisms causing latency spikes.
- *Interdependency mapping:* Understanding how the three root causes interact and why addressing them in the wrong order creates new problems.
- *Operational precision:* Recommendations with specific resource configurations, compliance dependencies, and implementation sequencing.
- *Constraint-aware scaling strategy:* A path to horizontal and vertical scaling that accounts for subnet capacity, SNAT allocation dynamics, and PCI compliance gates.
- *Cost awareness:* Leveraging existing reserved capacity rather than recommending new spend.

1.3 What CloudGo.ai Achieved

CloudGo.ai's context-aware analysis delivered the following outcomes:

1. Identified three converging root causes: CPU CFS throttling (pod-level compute starvation despite node headroom), SNAT port exhaustion (outbound connection ceiling on the Standard Load Balancer), and subnet IP exhaustion (Azure CNI address space blocking horizontal scale-out).
2. Demonstrated that the CTO's preferred horizontal scaling strategy was physically blocked by subnet capacity and would worsen SNAT exhaustion—a conclusion requiring synthesis of Container Insights telemetry, Load Balancer metrics, VNet IP allocation data, and PCI compliance policy.
3. Delivered a sequenced remediation plan that resolved immediate bottlenecks (NAT Gateway deployment and CFS quota adjustment) before enabling the scaling strategy, with explicit dependency chains and compliance gates.
4. Identified existing D16s_v5 Reserved Instances that could be applied during the node pool upgrade phase, eliminating incremental compute cost while doubling per-node capacity.
5. Provided detailed cost analysis with immediate versus future projections, checkpoint success metrics, and two concrete Terraform implementation paths.

2. Case Context

2.1 Company Profile: ReadyPay

ReadyPay is a Series B fintech company operating a real-time transaction processing platform on Azure Kubernetes Service. The platform handles payment authorizations and settlement for approximately 200 merchant integrations, processing transactions through upstream payment processors including StripeConnect, Adyen, and BankNet.

ReadyPay's technical footprint on Azure included:

1. Compute: A 3-node AKS cluster running Standard_D8s_v5 instances (8 vCPU, 32 GiB each) with Azure CNI networking on a /24 node subnet. The user pool ran 12 transaction-engine pods, 4 settlement workers, and 4 webhook processors under Guaranteed QoS class with CPU requests equal to limits (per Architecture Decision Record ADR-017).
2. Database: Azure SQL Database Premium P6 (800 DTU) for the transaction ledger, with geo-replication to Central US. A secondary Standard S3 instance for idempotency tracking.
3. Cache: Azure Cache for Redis Premium P1 with 3 shards and clustering enabled, accessed via Private Endpoint.
4. Network: Azure CNI networking with a /24 subnet (10.20.1.0/24), Standard Load Balancer with default outbound SNAT rules and 2 public IPs, Private Endpoints for SQL and Redis eliminating SNAT consumption for internal data calls.
5. Observability: Application Insights with Container Insights enabled, Log Analytics workspace for centralized telemetry.

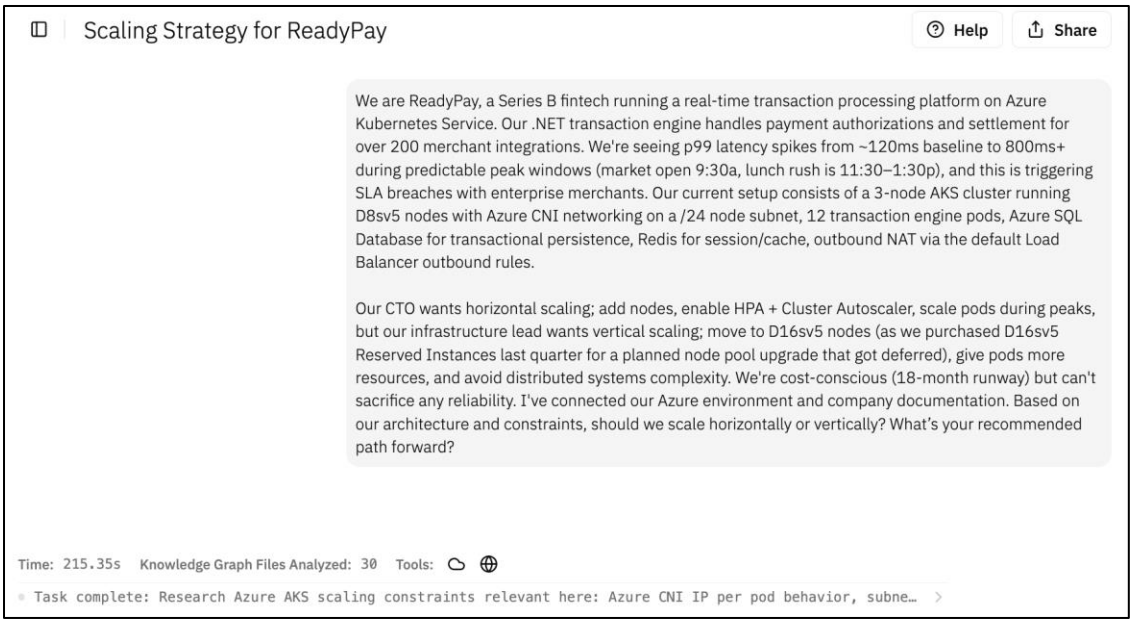
2.2 The Scaling Prompt

ReadyPay's engineering team was experiencing p99 latency spikes from approximately 120ms baseline to 800ms+ during predictable peak windows (9:30 AM EST market open, 11:30 AM to 1:30 PM EST lunch rush). These spikes were triggering SLA breaches with enterprise merchants. Leadership was split on the remediation approach.

The request to CloudGo.ai was framed as follows:

> *We are ReadyPay, a Series B fintech running a real-time transaction processing platform on Azure Kubernetes Service. Our .NET transaction engine handles payment authorizations and settlement for over 200 merchant integrations. We're seeing p99 latency spikes from ~120ms baseline to 800ms+ during predictable peak windows (market open 9:30a, lunch rush is 11:30–1:30p), and this is triggering SLA breaches with enterprise merchants. Our current setup consists of a 3-node AKS cluster running D8sv5 nodes with Azure CNI networking on a /24 node subnet, 12 transaction engine pods, Azure SQL Database for transactional persistence, Redis for session/cache, outbound NAT via the default Load Balancer outbound rules.*

Our CTO wants horizontal scaling; add nodes, enable HPA + Cluster Autoscaler, scale pods during peaks, but our infrastructure lead wants vertical scaling; move to D16sv5 nodes (as we purchased D16sv5 Reserved Instances last quarter for a planned node pool upgrade that got deferred), give pods more resources, and avoid distributed systems complexity. We're cost-conscious (18-month runway) but can't sacrifice any reliability. I've connected our Azure environment and company documentation. Based on our architecture and constraints, should we scale horizontally or vertically? What is your recommended path forward?

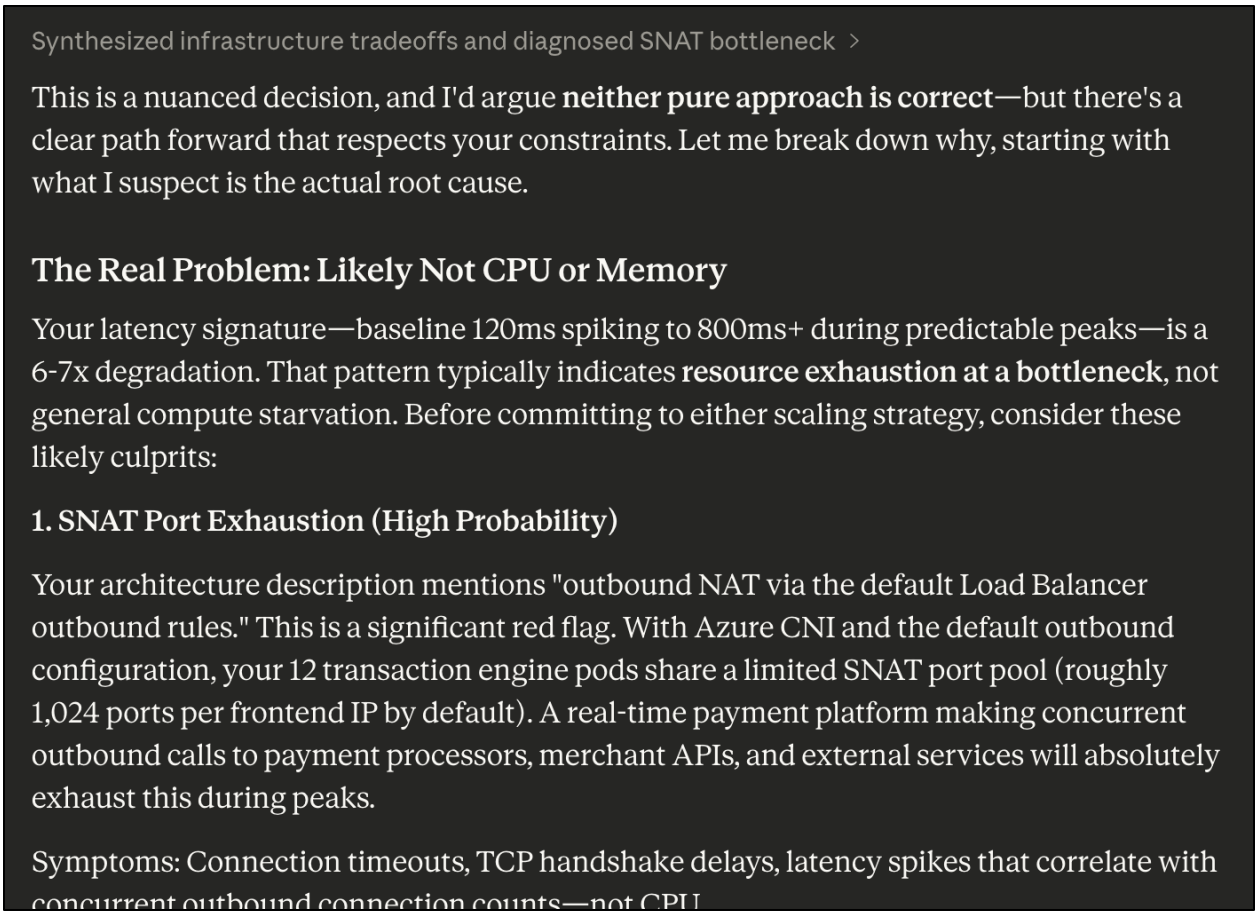


2.3 The Generic Response: What Claude Recommends Without Context

Before examining CloudGo.ai's context-aware analysis, it is instructive to understand what a capable LLM would recommend based on the prompt alone. We presented the identical prompt to Claude (Anthropic's frontier model) without any cloud infrastructure connection or company documentation. The response demonstrates sophisticated reasoning but necessarily operates on hypothesis rather than confirmation.

Claude's analysis began by correctly reframing the problem: *"This is a nuanced decision, and I'd argue neither pure approach is correct."* It identified SNAT port exhaustion as a *"high probability"* root cause, noting that the default Load Balancer outbound configuration was *"a significant red flag"* for a payment platform making concurrent outbound calls. Claude also flagged Azure SQL connection pool saturation and Redis connection multiplexing as possibilities worth investigating.

The recommended approach was strikingly similar in structure to what the full-context analysis would later confirm: deploy NAT Gateway first, then vertical scaling to leverage the Reserved Instances, then conservative horizontal scaling for resilience. Claude even provided specific Azure CLI commands for NAT Gateway deployment and an HPA configuration with sensible stabilization windows.



Synthesized infrastructure tradeoffs and diagnosed SNAT bottleneck >

This is a nuanced decision, and I'd argue **neither pure approach is correct**—but there's a clear path forward that respects your constraints. Let me break down why, starting with what I suspect is the actual root cause.

The Real Problem: Likely Not CPU or Memory

Your latency signature—baseline 120ms spiking to 800ms+ during predictable peaks—is a 6-7x degradation. That pattern typically indicates **resource exhaustion at a bottleneck**, not general compute starvation. Before committing to either scaling strategy, consider these likely culprits:

1. SNAT Port Exhaustion (High Probability)

Your architecture description mentions "outbound NAT via the default Load Balancer outbound rules." This is a significant red flag. With Azure CNI and the default outbound configuration, your 12 transaction engine pods share a limited SNAT port pool (roughly 1,024 ports per frontend IP by default). A real-time payment platform making concurrent outbound calls to payment processors, merchant APIs, and external services will absolutely exhaust this during peaks.

Symptoms: Connection timeouts, TCP handshake delays, latency spikes that correlate with concurrent outbound connection counts—not CPU

Claude's validation checklist demonstrated sound diagnostic thinking:

Check	How to Validate
SNAT exhaustion	Azure Monitor → Load Balancer → SNAT Connection Count; look for "Used SNAT Ports" approaching allocated
SQL connection queuing	Azure SQL → Query Performance Insight; check wait stats for RESOURCE_SEMAPHORE
Redis latency	Enable StackExchange.Redis profiling; check ConnectionMultiplexer.GetStatus() during peaks
Pod resource pressure	kubectl top pods during peak; if CPU isn't near limits, compute isn't your problem

This is genuinely useful guidance. It represents the quality of analysis a skilled cloud architect would provide during an initial consultation—before having access to the environment. The response correctly identifies the most likely culprits, proposes a reasonable remediation sequence, and provides a validation framework.

The critical distinction is not that Claude gets the direction wrong, but that Claude must recommend investigation where CloudGo.ai can provide confirmation. Claude hypothesizes that SNAT exhaustion is *"high probability"* and recommends checking the Load Balancer metrics.

CloudGo.ai's telemetry shows SNAT at 93% utilization with 45–47 connection failures per minute during the exact peak windows; the hypothesis is confirmed, the investigation is complete, and the remediation can begin immediately.

What Claude's response could not provide without context:

- Confirmation that CFS throttling is occurring at 3,200 events per minute (Claude suggests checking; CloudGo.ai confirms the exact number)
- Confirmation that SNAT is at 93% with 45 failures per minute (Claude flags as likely; CloudGo.ai provides the metrics)
- Confirmation that SQL is healthy at 78% DTU, eliminating it as a contributor (Claude suggests investigating; CloudGo.ai rules it out)
- Awareness that the /24 subnet has only 51 IPs remaining and horizontal scaling is physically blocked
- Reference to ADR-017's escalation criteria for CFS throttling thresholds
- Knowledge of the PCI compliance gate for subnet modifications or the 2-3 week lead time
- Connection to the March incident (INC-2025-031) where SNAT exhaustion was misdiagnosed as "network instability"

Decision: Horizontal vs Vertical (in your current environment)

Criterion	Horizontal scaling now	Vertical scaling now
Fixes pod CPU throttling from 2000m limit	No (limits still bind)	Only indirectly unless you also change limits; but gives more per-node headroom for fewer contention cases
Fixes SNAT exhaustion	Often worse (more pods/ connections)	Neutral; still need NAT Gateway
Feasible with /24 Azure CNI subnet nearly full	No (blocked)	Yes (but watch surge during upgrade)
Reliability risk	High	Medium (lower if you do it as controlled nodepool migration)
Cost/runway impact	Potentially higher (more nodes)	Favorable given you have unused D16s_v5 RIs

Therefore: *Vertical-first* (to capture RI value) **but only after** you address SNAT + throttling, and while you plan IP/network changes that make future horizontal scaling viable.

What I'd do in the next 30 days (crisp plan)

1. **Implement NAT Gateway** for AKS outbound (preserve existing public IPs if you have whitelisting requirements).
2. **Change transaction-engine CPU policy** (reduce CFS throttling): raise/remove CPU limits; verify with load test during peak windows.
3. **Plan/execute IP expansion or move to overlay/pod-subnet** so autoscaling is physically possible.
4. **Upgrade userpool to D16s_v5** to apply reserved capacity (but design the upgrade to avoid a surge node if IPs are still tight—this is a key rollout detail).
5. **Then** enable HPA + Cluster Autoscaler with conservative settings and guardrails.

3. Secure, Contextual Data Collection

The prompt above, while detailed, omits the constraints that determine whether a scaling strategy is operationally executable. CloudGo.ai's differentiation lies in its ability to gather and synthesize context that traditional analysis cannot access. This context comes from two sources: cloud infrastructure connections and company documentation integration.

3.1 Cloud Infrastructure Connection

CloudGo.ai connects to customer cloud environments via read-only CLI credentials (Azure CLI, AWS CLI, GCP gcloud). This connection enables infrastructure inventory scans, usage telemetry retrieval, cost and billing data access, and resource configuration analysis.

For ReadyPay, the Azure scan surfaced the following critical information:

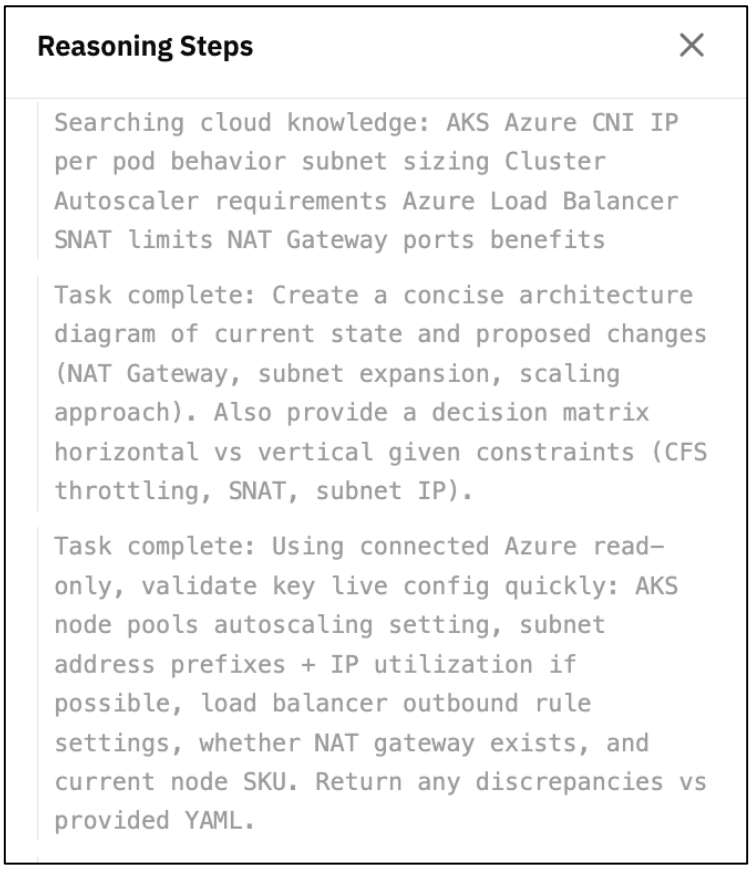
Data Point	Finding
Pod CPU CFS Throttling	~3,200 throttled periods/min peak despite 58% avg node CPU
SNAT Port Utilization	~93% peak (approaching 1,024 port allocation ceiling)
SNAT Connection Failures	~45–47 failures/min during peak traffic windows
Subnet IP Remaining	51 of 256 addresses in /24 subnet; 0 additional nodes possible
SQL DTU Utilization	78.3% peak — healthy, NOT the bottleneck
Redis Server Load	0.38 avg, 0.72 max — healthy
D16s_v5 Reserved Instances	3 instances purchased, planned upgrade deferred pending stabilization

3.2 Company Documentation Integration

CloudGo.ai ingests customer-provided documentation into a secure, private document store. Documents are indexed for semantic retrieval during analysis. Each customer's documentation exists in a private collection with no cross-tenant data access. For ReadyPay, the following documents were ingested and queried:

Document	Key Information Retrieved
ADR-017 (Pod Resource Limits)	Guaranteed QoS with 2000m CPU limits mandated after May 2025 Redis noisy-neighbor incident. Escalation criteria: investigate if throttle rate exceeds 500/min
INC-2025-031 (March Incident)	2.3% outbound API timeout rate during peak. Attributed to "network instability" — Azure support suggested SNAT investigation. NAT Gateway action item opened, never pursued

Document	Key Information Retrieved
PCI Network Change Policy	Subnet CIDR changes classified as Tier 1 network changes. Required: security review (3–5 days) + compliance sign-off (2–3 days). Total lead time: 2–3 weeks
Vendor API Requirements	Three payment processors require IP whitelisting. BankNet requires 30-day notice for IP changes. Existing public IPs can be reassociated to NAT Gateway
Team Skills Assessment	4-person infrastructure team. Strong AKS/SQL expertise, limited Azure networking depth. External networking consultant contract ended August 2025
RI Cost Report	3× D16s_v5 Reserved Instances purchased September 2025. Node pool upgrade deferred pending latency investigation



4. Synthesis: How CloudGo.ai Reasons Over Multi-Source Context

The same natural language prompt is now executed with the Azure CLI connection active (cloud scan and usage data available), the company documentation collection linked (ADR-017, incident postmortem, PCI policy, vendor requirements, skills assessment, cost reports), and CloudGo.ai's full tooling stack available for research and execution planning.

CloudGo.ai's opening assessment immediately reframed the question: *"Horizontal scaling is not the right first move. You have two non-node-capacity bottlenecks that will continue to trigger p99 spikes (or get worse) even if you add nodes/pods."* The response identified all three root causes in the first paragraph, with specific metrics from the telemetry confirming each one.

4.1 CloudGo.ai's Unique Tooling Stack

CloudGo.ai brings multiple resources to bear on the augmented prompt. The curated cloud knowledge base contains documentation for AWS, Azure, and GCP including Well-Architected Frameworks, service-specific FAQs and pricing documentation, operational best practices, and Kubernetes resource management references. The response included direct links to Microsoft documentation for NAT Gateway SNAT port scaling, Load Balancer outbound connection troubleshooting, and AKS IP address planning guidance.

CloudGo.ai also executes read-only CLI commands against connected cloud environments, retrieving real-time infrastructure state, usage metrics, and cost data. This validates assumptions against actual resource configurations rather than relying on user-reported descriptions. The platform further maintains a searchable index of Terraform provider documentation and module configurations, enabling generation of infrastructure-as-code scaffolding for recommended changes.

4.2 Synthesis in Action: CloudGo.ai's Reasoning Process

CloudGo.ai recognizes this as an AKS performance optimization problem requiring both infrastructure telemetry analysis and organizational context retrieval. The reasoning process demonstrates multi-source synthesis at each stage of the diagnosis.

The first synthesis moment occurs when CloudGo.ai examines the Container Insights data and confirms severe CFS throttling: *"transaction-engine pods are pinned at a 2000m CPU limit and are hitting that ceiling during peaks... throttling spikes to ~3,200 events/min."* This is not a hypothesis to investigate; it is a confirmed finding. CloudGo.ai then connects this to ADR-017, which mandated the aggressive limits after a noisy-neighbor incident, and notes that the observed throttling rate exceeds ADR-017's own escalation threshold by more than six times.

The second synthesis moment connects Load Balancer SNAT metrics to the outbound connection pattern: *"Used SNAT Ports peaks ~93% with connection failures ~45–47/min in the same peak windows."* CloudGo.ai explicitly warns that scaling out pods would worsen this: *"Scaling out pods typically increases outbound concurrency and can increase SNAT pressure, causing more dependency timeouts and latency."* When ReadyPay confirmed that their .NET

services open many short-lived connections per transaction, CloudGo.ai added app-level recommendations for connection pooling alongside the NAT Gateway deployment.

The third synthesis moment is the one entirely absent from Claude's response: CloudGo.ai connects Azure CNI IP allocation to the subnet capacity constraint. *"Your Azure CNI /24 subnet has no node scale headroom... your scan shows you can't add even one surge node safely. This means Cluster Autoscaler can't work reliably until you fix IP capacity."* The CTO's preferred approach is not merely inadvisable, it is physically blocked by network topology. CloudGo.ai further identifies that fixing this constraint requires navigating ReadyPay's PCI-DSS Tier 1 change control process, establishing a 2-3 week compliance lead time before horizontal scaling becomes possible.

The convergence of these three findings is what makes multi-source synthesis transformative. No single data source—not the Container Insights dashboard, not the Load Balancer metrics, not the VNet configuration, not the PCI policy—contains the complete picture. A monitoring dashboard might surface one or two of these signals. Only synthesis across all data sources reveals the three-problem convergence and its implications for remediation sequencing.

5. CloudGo.ai's Comprehensive Action Plan

5.1 Strategic Recommendation

CloudGo.ai's synthesized recommendation reframed the scaling debate entirely: *"The correct answer isn't 'horizontal vs vertical' as a single decision—it's sequence the fixes so you can safely do either (or both) without making reliability worse."* The recommended path forward follows a strict dependency chain with clear success checkpoints:

Phase 0 (this week): Stop the two real causes of p99 spikes. Deploy NAT Gateway for AKS egress, preserving existing public IPs for payment processor whitelisting. NAT Gateway provides 64,512 SNAT ports per public IP (versus 1,024 in the default allocation) and can scale by attaching additional IPs. Simultaneously, raise or remove CPU limits for transaction-engine while maintaining CPU requests, a common pattern for latency-sensitive services that should not be throttled during burst.

Phase 1 (weeks 1–3): Unblock safe scaling by fixing IP constraints. With Cluster Autoscaler and even node-pool upgrades with surge blocked by the /24 subnet, ReadyPay must expand the CIDR, add additional subnets with a new node pool, or migrate to Azure CNI Overlay or Dynamic IP Allocation. CloudGo.ai noted that networking changes are *"operationally heavy in regulated fintech environments,"* acknowledging the PCI/Tier-1 change controls while establishing this as the critical path for enabling the CTO's desired scaling posture.

Phase 2 (afterwards): Apply vertical scaling to capture RI value, then add horizontal scaling. Once SNAT and throttling are resolved and IP headroom exists, upgrade the user pool from D8s_v5 to D16s_v5 to apply the existing Reserved Instances. Then enable HPA and Cluster Autoscaler with conservative bounds, only after the prerequisites are in place.

5.2 The Decision Framework

CloudGo.ai provided a decision matrix comparing horizontal versus vertical scaling against ReadyPay's specific constraints:

Criterion	Horizontal Now vs. Vertical Now
Fixes pod CPU throttling	Horizontal: No (limits still bind) | Vertical: Only indirectly; more per-node headroom
Fixes SNAT exhaustion	Horizontal: Often worse (more pods/connections) | Vertical: Neutral; still need NAT Gateway
Feasible with /24 nearly full	Horizontal: No (blocked) | Vertical: Yes (but watch surge during upgrade)
Reliability risk	Horizontal: High | Vertical: Medium (lower with controlled migration)
Cost/runway impact	Horizontal: Higher (more nodes) | Vertical: Favorable (unused D16s_v5 RIs)

The conclusion: Vertical-first (to capture RI value) but only after you address SNAT + throttling, and while you plan IP/network changes that make future horizontal scaling viable.

5.3 Cost Analysis: Immediate vs. Future

CloudGo.ai provided detailed cost projections split into immediate changes (next 1–2 weeks) and future changes (1–2 quarters), enabling ReadyPay to budget for the remediation while understanding the reliability tradeoffs.

Immediate Changes (Next 1–2 Weeks)

Change	Cost Impact	Notes
Azure NAT Gateway	Increase (moderate)	Bills per hour + data processed; reliability gain typically worth it for fintech SLAs
Additional Public IPs for NAT GW	Increase (low)	Each IP adds 64,512 SNAT ports; most teams start with 2 IPs
.NET connection pooling fixes	Near-zero	Engineering time only; reduces compute + network overhead
Raise/remove CPU limits	Near-zero	May see higher instantaneous CPU; no direct Azure bill change

Future Changes (1–2 Quarters)

Change	Cost Impact	Notes
Subnet expansion / overlay / dynamic IP	Depends	Mostly engineering + change-management cost; reduces future emergency overprovisioning
Upgrade to D16s_v5 using existing RIs	Neutral or decrease	Captures reserved capacity value; stops wasting unused RI amortization
HPA + Cluster Autoscaler	Variable (can increase)	Scale-out during peaks increases cost unless aggressive scale-down; safe after NAT + connection reuse

CloudGo.ai summarized the net expectation: *"Slight infra cost increase (NAT), but material reliability improvement and likely reduction in merchant-facing tail latency."*

2) Cost analysis (immediate vs future)

You asked for "immediate and future cost analysis." I'll give you a **component-based model** with what tends to dominate cost, and where you should expect net increases/decreases. (I can tighten this with exact numbers once I can validate your Azure pricing region + egress volume + current node counts from the live account.)

Immediate changes (next 1–2 weeks)

Change	Immediate cost delta	Why / notes
Azure NAT Gateway	**Increase** (moderate)	NAT Gateway bills per hour and per data processed; plus any attached public IPs/prefix. See pricing page for exact rates in eastus2: NAT Gateway pricing. Reliability gain is usually worth it for fintech SLAs.
Add **1–N Public IPs** (or a prefix) to NAT GW	**Increase** (low → moderate)	Each additional IP adds another **64,512 SNAT ports** (SNAT scaling). Many teams start with 2 IPs, then scale if needed.
.NET connection pooling / keep-alives	**Near-zero**	Engineering time only. Often reduces compute + network overhead, and reduces error rates.
Raise/remove CPU limits	**Near-zero** (but watch node saturation)	You may see higher instantaneous CPU usage; usually no direct Azure bill change unless you scale nodes. Biggest "cost" is the need to ensure proper requests/priority and possibly isolate workloads.
NAT GW monitoring/flow logs (optional)	**Small increase**	Flow logs can add a small fixed charge plus Log Analytics ingestion; see flow logs FAQ (flow logs cost note).

Net immediate expectation: Slight infra cost increase (NAT), but **material reliability improvement** and likely reduction in merchant-facing tail lat.

Future changes (1–2 quarters)

Change	Future cost delta	Why / notes
Subnet expansion / overlay / dynamic IP	Depends	Mostly engineering + change-management cost; can reduce future need for emergency node overprovisioning. Overlay/dynamic IP can avoid needing huge VNets just to scale pods.
Vertical upgrade to D16s_v5 using existing RIs	Likely **neutral or decrease vs today** (if you're currently paying on-demand D8s)	Since you already own D16s reservations, the key is to **actually apply them**. Your scan suggests substantial savings vs on-demand (and you stop wasting unused RI amortization).
HPA + Cluster Autoscaler	Variable (can increase)	If you scale out during peaks, cost goes up unless you scale back down aggressively. After NAT + connection reuse, autoscaling becomes safe and predictable. AKS cluster autoscaler behavior is documented here: AKS cluster autoscaler.
Add more NAT GW IPs (only if needed)	Low → moderate	Cheap reliability headroom: each IP adds a lot of SNAT ports.
Consider egress control (Azure Firewall + NAT)	Increase	If you later need strict egress policy + inspection, Firewall adds cost; you can integrate it with NAT for SNAT scale (Firewall + NAT).

5.4 Implementation Roadmap with Success Checkpoints

CloudGo.ai delivered a concrete week-by-week implementation plan with measurable success criteria:

Week 1:

- Deploy NAT Gateway + 2 outbound IPs (or prefix)
- Add alerts on NAT failures and connection counts
- Ship .NET connection reuse fixes behind a feature flag
- Raise/remove CPU limits for transaction-engine; confirm throttling drops

Checkpoint success metrics:

- SNAT failures → near zero
- Dependency p99 (processor/webhook) drops materially
- Transaction-engine CPU throttling reduced dramatically
- Overall request p99 moves back toward baseline

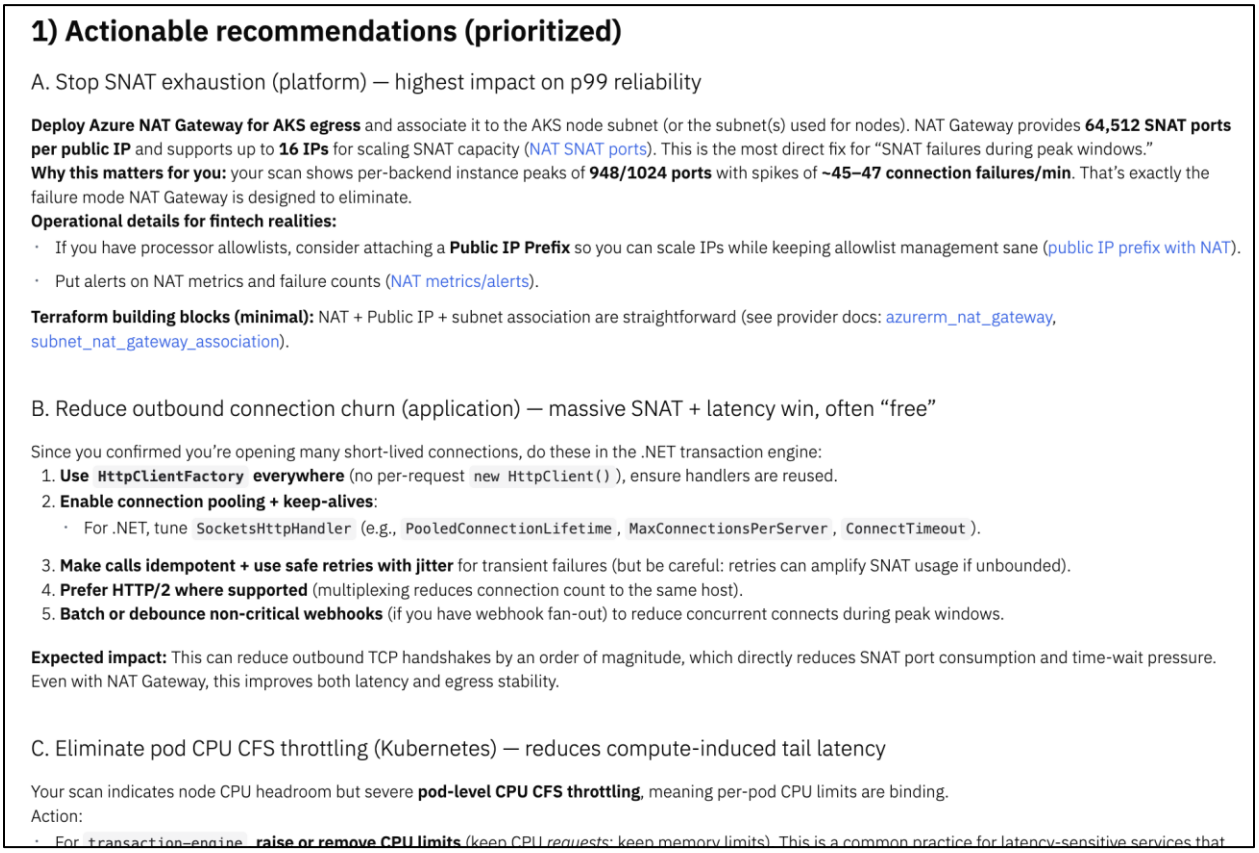
Weeks 2–4:

- Choose subnet strategy (expand vs overlay vs dynamic IP)
- Execute node pool migration accordingly

After/Month 2:

- Upgrade userpool to D16s_v5 to apply reservations (ensure IP headroom for surge)
- Enable HPA conservatively (scale on RPS and CPU requests)

- Enable Cluster Autoscaler only after subnet/IP is solved and test failure modes



1) Actionable recommendations (prioritized)

A. Stop SNAT exhaustion (platform) — highest impact on p99 reliability

Deploy Azure NAT Gateway for AKS egress and associate it to the AKS node subnet (or the subnet(s) used for nodes). NAT Gateway provides **64,512 SNAT ports per public IP** and supports up to **16 IPs** for scaling SNAT capacity (NAT SNAT ports). This is the most direct fix for "SNAT failures during peak windows."
Why this matters for you: your scan shows per-backend instance peaks of **948/1024 ports** with spikes of **~45–47 connection failures/min**. That's exactly the failure mode NAT Gateway is designed to eliminate.
Operational details for fintech realities:

- If you have processor allowlists, consider attaching a **Public IP Prefix** so you can scale IPs while keeping allowlist management sane (public IP prefix with NAT).
- Put alerts on NAT metrics and failure counts (NAT metrics/alerts).

Terraform building blocks (minimal): NAT + Public IP + subnet association are straightforward (see provider docs: azurerm_nat_gateway, subnet_nat_gateway_association).

B. Reduce outbound connection churn (application) — massive SNAT + latency win, often "free"

Since you confirmed you're opening many short-lived connections, do these in the .NET transaction engine:

1. **Use `HttpClientFactory` everywhere** (no per-request `new HttpClient()`), ensure handlers are reused.
2. **Enable connection pooling + keep-alives**:
   - For .NET, tune `SocketsHttpHandler` (e.g., `PooledConnectionLifetime`, `MaxConnectionsPerServer`, `ConnectTimeout`).
3. **Make calls idempotent + use safe retries with jitter** for transient failures (but be careful: retries can amplify SNAT usage if unbounded).
4. **Prefer HTTP/2 where supported** (multiplexing reduces connection count to the same host).
5. **Batch or debounce non-critical webhooks** (if you have webhook fan-out) to reduce concurrent connects during peak windows.

Expected impact: This can reduce outbound TCP handshakes by an order of magnitude, which directly reduces SNAT port consumption and time-wait pressure. Even with NAT Gateway, this improves both latency and egress stability.

C. Eliminate pod CPU CFS throttling (Kubernetes) — reduces compute-induced tail latency

Your scan indicates node CPU headroom but severe **pod-level CPU CFS throttling**, meaning per-pod CPU limits are binding.
Action:

- For `transaction-engine`, **raise or remove CPU limits** (keep CPU *requests*; keep memory limits). This is a common practice for latency-sensitive services that

5.5 Costed Implementation Plan

When asked to generate a costed implementation plan, CloudGo.ai produced a phase-by-phase cost model with explicit formulas for each component. The response demonstrated both the depth of analysis possible with connected infrastructure and appropriate epistemic honesty about what data was needed for precise calculations.

CloudGo.ai then produced a "current vs. post-change" monthly delta table, computing the net financial impact of the full remediation plan:

Item	Current	After Plan / Delta
D16s_v5 reservation (amortized)	$2,460/mo	$2,460/mo (already paying)
On-demand compute (eliminable)	+$1,960/mo	−$1,960/mo
NAT Gateway fixed hourly	$0	+~$32–45/mo
NAT Gateway data processed	$0	+variable (needs GB/mo)
Public IPs for NAT (2 recommended)	$0	+~$15–20/mo
Net monthly delta		Likely net-neutral or net-saving

The conclusion: the remediation plan is *"very likely net-cost-neutral or net-saving after the D16 migration, even with NAT Gateway."* The $1,960/month savings from applying the existing Reserved Instances more than offsets the NAT Gateway fixed costs. Variable data processing charges depend on actual egress volume, which CloudGo.ai offered to compute by pulling metrics from the connected Azure environment.

This cost analysis demonstrates a key differentiator of context-aware infrastructure intelligence: the ability to compute financial impact from actual resource configurations and billing data, rather than providing generic pricing guidance that requires the customer to assemble the inputs themselves.

5.6 Execution-Ready Implementation Guidance

CloudGo.ai's value extends beyond diagnosis and planning into execution support. When asked for implementation details, CloudGo.ai produced a comprehensive runbook with copy-paste CLI commands, Kubernetes manifests, and .NET code snippets, all tailored to ReadyPay's specific resource names, regions, and configurations.

Measuring actual egress volume. CloudGo.ai provided Azure CLI commands to query 30-day outbound bytes from the AKS node NICs, enabling precise NAT Gateway cost projection:

1C) Query outbound bytes per NIC for last 30 days

Azure Monitor metric commonly available on NICs is **Bytes Sent**.

1. Get NIC IDs:

```
NIC_IDS=$(az network nic list -g "$NODE_RG" --query "[].id" -o tsv)
echo "$NIC_IDS" | wc -l
```

2. For each NIC, query bytes sent and sum. Here's a bash + jq approach:

```
START=$(date -u -d "30 days ago" +"%Y-%m-%dT%H:%M:%SZ")
END=$(date -u +"%Y-%m-%dT%H:%M:%SZ")

TOTAL_BYTES=0

for NIC in $NIC_IDS; do
  BYTES=$(az monitor metrics list \
    --resource "$NIC" \
    --metric "BytesSent" \
    --interval PT1H \
    --start-time "$START" \
    --end-time "$END" \
    --aggregation Total \
    --query "value[0].timeseries[0].data[].total | sum(@)" -o tsv 2>/dev/null)

  if [[ -n "$BYTES" ]]; then
```

Deploying NAT Gateway with stable IPs. The runbook included complete Azure CLI commands

for creating a NAT Gateway with a public IP prefix (recommended for allowlisting), associating it to the AKS subnet, and validating the configuration:

2) Deploy NAT Gateway with stable outbound IPs (CLI)

2A) Create a Public IP Prefix (recommended for allowlisting)

A prefix makes it easier to scale IP count later while keeping addresses contiguous.

```
LOCATION="eastus2"
NAT_PIP_PREFIX="rp-prod-nat-pip-prefix"
NAT_PREFIX_LEN=29  # /29 gives 8 IPs (5 usable typically). Choose based on allowlist needs.

az network public-ip prefix create \
  -g "$RG" -n "$NAT_PIP_PREFIX" \
  -l "$LOCATION" \
  --prefix-length "$NAT_PREFIX_LEN" \
  --sku Standard
```

If you prefer just 2 discrete IPs (simpler):

```
az network public-ip create -g "$RG" -n rp-prod-nat-pip-01 -l "$LOCATION" --sku Standard --allocation-method Static
az network public-ip create -g "$RG" -n rp-prod-nat-pip-02 -l "$LOCATION" --sku Standard --allocation-method Static
```

2B) Create NAT Gateway

```
NAT="rp-prod-natgw"

# If using IP prefix:
az network nat gateway create \
  -g "$RG" -n "$NAT" -l "$LOCATION" \
  --public-ip-prefixes "$NAT_PIP_PREFIX" \
  --idle-timeout 10

# If using discrete Public IPs:
# az network nat gateway create \
#   -g "$RG" -n "$NAT" -l "$LOCATION" \
#   --public-ip-addresses rp-prod-nat-pip-01 rp-prod-nat-pip-02 \
#   --idle-timeout 10
```

.NET connection pooling configuration. Production-ready C# code for HttpClientFactory with SocketsHttpHandler to reduce SNAT pressure from short-lived connections:

5A) Use HttpClientFactory + SocketsHttpHandler tuning

In `Program.cs` / DI setup:

```
using System.Net;
using System.Net.Http;

builder.Services.AddHttpClient("ProcessorClient")
    .ConfigurePrimaryHttpMessageHandler(() => new SocketsHttpHandler
    {
        AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate,
        PooledConnectionLifetime = TimeSpan.FromMinutes(5),   // recycle to respect DNS/LB changes
        PooledConnectionIdleTimeout = TimeSpan.FromMinutes(2),
        MaxConnectionsPerServer = 200,                        // tune based on peak concurrency per pod
        ConnectTimeout = TimeSpan.FromSeconds(2),
        KeepAlivePingDelay = TimeSpan.FromSeconds(30),        // for HTTP/2 scenarios
        KeepAlivePingTimeout = TimeSpan.FromSeconds(15),
        EnableMultipleHttp2Connections = true                 // if you use HTTP/2 and need more parallelism
    })
    .SetHandlerLifetime(Timeout.InfiniteTimeSpan); // don't rotate handlers unless you want to
```

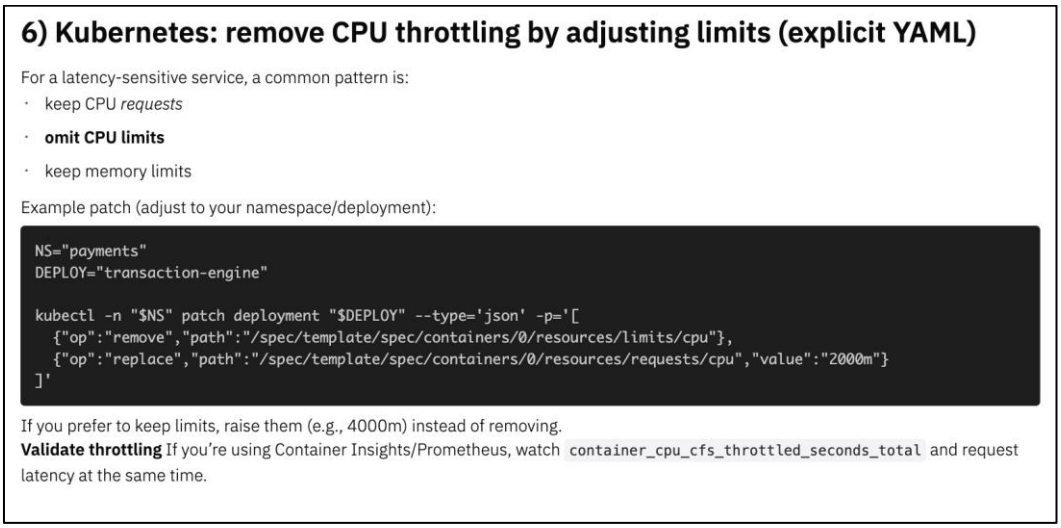
Usage:

```
public class ProcessorGateway
{
    private readonly IHttpClientFactory _factory;

    public ProcessorGateway(IHttpClientFactory factory) => _factory = factory;

    public async Task<HttpResponseMessage> AuthorizeAsync(HttpRequestMessage req, CancellationToken ct)
    {
        var client = _factory.CreateClient("ProcessorClient");
        return await client.SendAsync(req, HttpCompletionOption.ResponseHeadersRead, ct);
    }
}
```

Kubernetes CPU limit removal. For the CFS throttling fix, CloudGo.ai provided a kubectl patch command to remove CPU limits while preserving requests, the recommended pattern for latency-sensitive services:



6) Kubernetes: remove CPU throttling by adjusting limits (explicit YAML)

For a latency-sensitive service, a common pattern is:

- keep CPU *requests*
- **omit CPU limits**
- keep memory limits

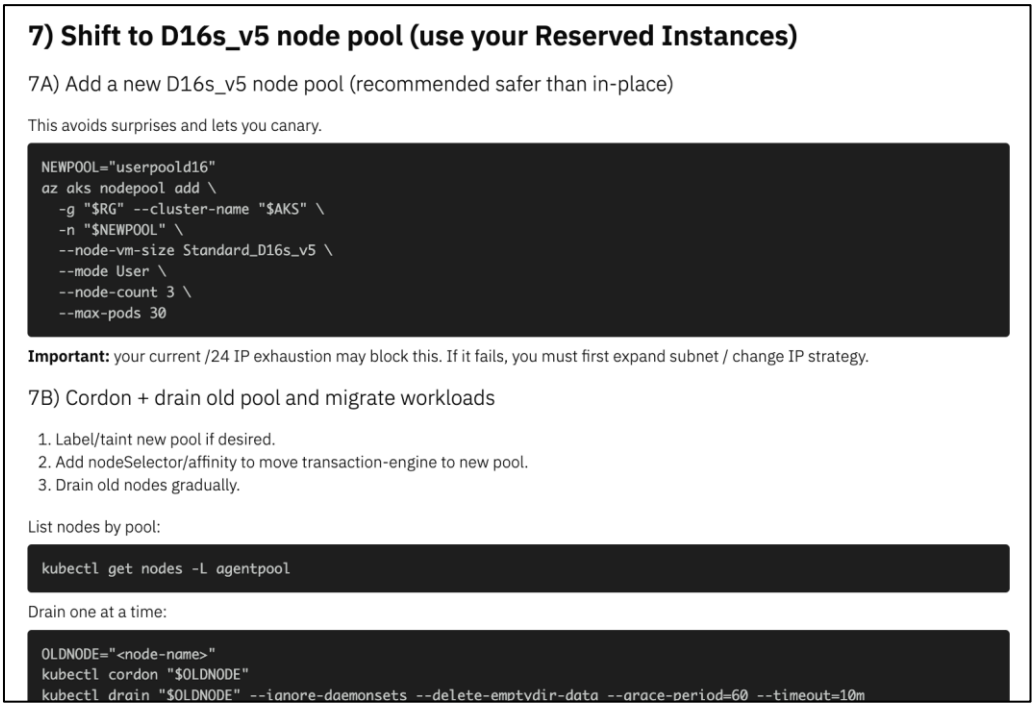
Example patch (adjust to your namespace/deployment):

```
NS="payments"
DEPLOY="transaction-engine"

kubectl -n "$NS" patch deployment "$DEPLOY" --type='json' -p='[
  {"op":"remove","path":"/spec/template/spec/containers/0/resources/limits/cpu"},
  {"op":"replace","path":"/spec/template/spec/containers/0/resources/requests/cpu","value":"2000m"}
]'
```

If you prefer to keep limits, raise them (e.g., 4000m) instead of removing.
Validate throttling If you're using Container Insights/Prometheus, watch `container_cpu_cfs_throttled_seconds_total` and request latency at the same time.

Node pool migration to D16s_v5. The runbook included the complete workflow for migrating to the Reserved Instance SKU: create new pool, cordon and drain old nodes, validate workload migration, then delete the old pool. CloudGo.ai noted the /24 IP constraint and warned that the nodepool add command would fail if subnet capacity wasn't addressed first.



7) Shift to D16s_v5 node pool (use your Reserved Instances)

7A) Add a new D16s_v5 node pool (recommended safer than in-place)

This avoids surprises and lets you canary.

```
NEWPOOL="userpoold16"
az aks nodepool add \
  -g "$RG" --cluster-name "$AKS" \
  -n "$NEWPOOL" \
  --node-vm-size Standard_D16s_v5 \
  --mode User \
  --node-count 3 \
  --max-pods 30
```

Important: your current /24 IP exhaustion may block this. If it fails, you must first expand subnet / change IP strategy.

7B) Cordon + drain old pool and migrate workloads

1. Label/taint new pool if desired.
2. Add nodeSelector/affinity to move transaction-engine to new pool.
3. Drain old nodes gradually.

List nodes by pool:

```
kubectl get nodes -L agentpool
```

Drain one at a time:

```
OLDNODE="<node-name>"
kubectl cordon "$OLDNODE"
kubectl drain "$OLDNODE" --ignore-daemonsets --delete-emptydir-data --grace-period=60 --timeout=10m
```

This level of execution detail transforms CloudGo.ai from an advisory tool into an implementation accelerator. ReadyPay's infrastructure team could execute Phase 0 (NAT Gateway + CPU limits) within hours of receiving recommendations, with confidence that commands are tailored to their specific environment rather than generic examples.

6. Conclusion

6.1 The Transformation in Infrastructure Recommendations

The infrastructure optimization recommendation changed fundamentally when organizational context was incorporated. The following table summarizes the contrast between Claude's prompt-only analysis and CloudGo.ai's context-aware response:

Claude (Without Context)	CloudGo.ai (With Context)
SNAT exhaustion is "high probability"	SNAT at 93% with 45 failures/min — confirmed root cause
Suggests checking for CPU throttling	CFS throttling at 3,200/min; 6× ADR-017's escalation threshold
Suggests investigating SQL connection queuing	SQL at 78% DTU, Redis at 0.38 load — eliminated as causes
No awareness of subnet constraint	/24 has 51 IPs remaining; horizontal scaling physically blocked
Recommends checking SNAT metrics	Provides exact peak utilization and failure rate with correlation to latency spikes
Generic phasing ("this week" / "next 2 weeks")	Sequenced plan with explicit dependency chain, PCI compliance gates, and checkpoint metrics
Suggests using Reserved Instances	Confirms RI availability and sequences into upgrade after prerequisites
Provides CLI commands for NAT Gateway	Provides two Terraform implementation paths with tradeoff analysis for allowlisting requirements
No awareness of March incident	Connects current symptoms to INC-2025-031 misdiagnosis

Infrastructure optimization fails when analysis operates in isolation from configuration history, compliance constraints, and operational context. The information required for sound optimization decisions exists within every organization. It resides in architecture decision records, incident postmortems, compliance policy documents, and the cost management systems that track reserved capacity. The challenge is not generating recommendations. The challenge is gathering and synthesizing the context that makes recommendations executable.

6.2 A Different Kind of Infrastructure Intelligence

The Claude response was not wrong. It was genuinely helpful, directionally correct, and demonstrated sophisticated reasoning about the problem space. A team following Claude's validation checklist would eventually arrive at the same root causes. The difference is time and

certainty.Claude's response requires ReadyPay to investigate before acting. Check the SNAT metrics. Profile the Redis connections. Run kubectl top during peak. Validate each hypothesis before committing to remediation. This is the appropriate recommendation when you don't have access to the data.

CloudGo.ai's response arrives pre-validated: the SNAT metrics show 93% utilization and 45 failures per minute. The CPU throttling has been measured at 3,200 events per minute against a 500/min escalation threshold. The SQL has been found to be healthy at 78% DTU and can be ruled out. For a fintech experiencing SLA breaches with enterprise merchants, this difference is not academic. Every week spent validating hypotheses is another week of latency spikes, another week of merchant complaints, another week of commercial risk. Context-aware analysis compresses the timeline from "investigate then remediate" to "remediate now with these specific parameters."

The ReadyPay case illustrates a broader truth about infrastructure optimization: the constraint that determines success or failure is rarely visible in the monitoring dashboard. It is configurational (the CFS quota from ADR-017), architectural (the SNAT allocation model of the default Load Balancer), topological (the /24 subnet's IP ceiling), procedural (the PCI compliance gate), or financial (the unused Reserved Instances). Tools that address only the telemetry dimension produce recommendations that are correct in the abstract but incomplete in the concrete reality of operational execution.

Context-aware infrastructure intelligence does not require omniscience. It requires the ability to gather information from where it lives, synthesize across organizational boundaries, and reason about constraints that traditional monitoring tools ignore. The result is not a different answer to the same question, but a different question entirely: not "should we scale horizontally or vertically?" but "what prerequisites must be resolved before either scaling strategy can be safely executed, and in what order?" The answer to that question is immediately actionable. The answer to the first question is a debate that can cycle indefinitely while p99 latency continues to spike.