Azure SME

Cross Roles

- Architecture & Strategy
- Leadership & influence
- Problem solving

Architecture & strategy

1 Evaluating cost vs performance vs security

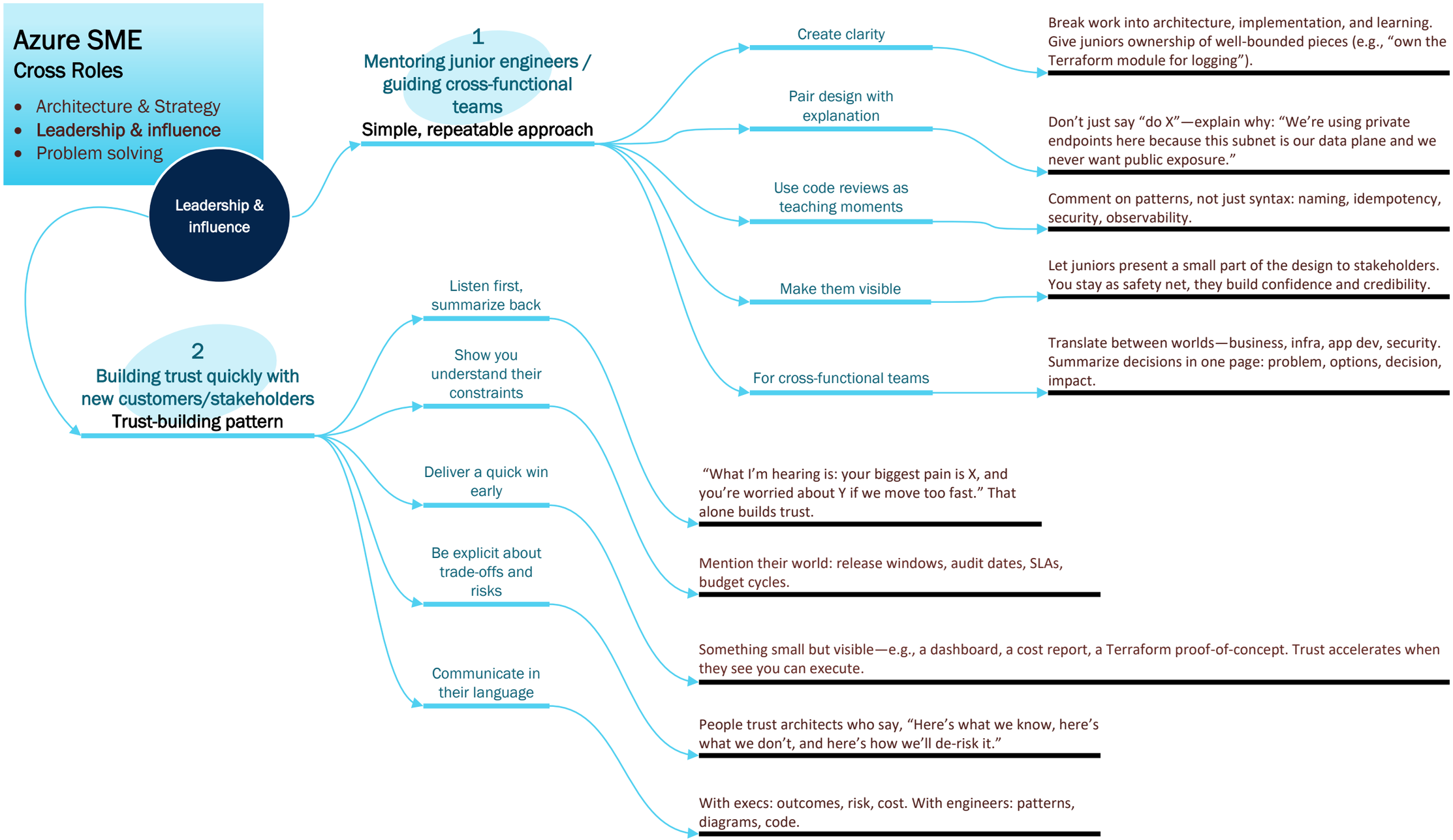
General approach

- Business outcomes — **What matters most ?**
  - Is this **workload** revenue-critical, compliance-sensitive, or cost-sensitive
    Rank: Availability, security, cost, time-to-market
- SLOs and constraints — **Let's put numbers on it …**
  - Uptime target, RPO/RTO, latency expectations, data residency, regulatory requirements.
    These become my guardrails.
- Design Options — Never a single Answer — **I'll offer 2-3 patterns here**
  - **Option A**: Premium PaaS, ZRS, private endpoints, full Defender—high cost, high resilience.
    Option B: Standard SKUs, zone-redundant only for critical components—balanced.
    Option C: Single region, minimal redundancy—low cost, higher risk.
- Make trade-offs explicit — **Show the deltas.**
  - For each option: cost estimate, risk profile, operational complexity. I'll use language like: "For ~20% more cost, you get zone redundancy and 50% lower risk of regional outage impact."
- Default to secure by design — **Security is not optional.**
  - Private endpoints, managed identities, encryption, basic policies are non-negotiable; you trade mostly between performance and cost, not security and cost.

2 High-impact decision with incomplete information

Scenario pattern to use

- Situation
  - A customer needed to move a critical workload to Azure under a tight deadline, but usage patterns and future scale were unclear.
- Tension
  - Had to choose between
    A - Fully containerized microservices on AKS (flexible, complex), or
    B - Lift-and-shift to App Service/VMs with some modernization later.
- Decision pattern
  - Chose a "minimum regret" architecture:
    - Use App Service + managed DB + standard networking patterns.
    - Design the solution so that future containerization is possible (clean boundaries, externalized config, managed identity, etc.).
    - Documented what assumptions you were making (expected load, growth, release cadence).
- Why it works in architectural design discussions ?
  - Shows that when information is incomplete, you:
    - Optimize for reversibility
    - Avoid over-engineering
    - Make assumptions explicit
    - Leave a clear path to evolve the architecture


Azure SME
Cross Roles
Architecture & Strategy
Leadership & influence
Problem solving
Leadership & influence
1
Mentoring junior engineers / guiding cross-functional teams
Simple, repeatable approach
Create clarity
Break work into architecture, implementation, and learning. Give juniors ownership of well-bounded pieces (e.g., "own the Terraform module for logging").
Pair design with explanation
Don't just say "do X"—explain why: "We're using private endpoints here because this subnet is our data plane and we never want public exposure."
Use code reviews as teaching moments
Comment on patterns, not just syntax: naming, idempotency, security, observability.
Make them visible
Let juniors present a small part of the design to stakeholders. You stay as safety net, they build confidence and credibility.
For cross-functional teams
Translate between worlds—business, infra, app dev, security. Summarize decisions in one page: problem, options, decision, impact.
2
Building trust quickly with new customers/stakeholders
Trust-building pattern
Listen first, summarize back
"What I'm hearing is: your biggest pain is X, and you're worried about Y if we move too fast." That alone builds trust.
Show you understand their constraints
Mention their world: release windows, audit dates, SLAs, budget cycles.
Deliver a quick win early
Something small but visible—e.g., a dashboard, a cost report, a Terraform proof-of-concept. Trust accelerates when they see you can execute.
Be explicit about trade-offs and risks
People trust architects who say, "Here's what we know, here's what we don't, and here's how we'll de-risk it."
Communicate in their language
With execs: outcomes, risk, cost. With engineers: patterns, diagrams, code.

Azure SME

Cross Roles

- Architecture & Strategy
- Leadership & influence
- **Problem solving**

Problem solving

1
A technically complex Azure problem & how to approach it

Example problem: A multi-region, microservices-based app with Event Hubs, Service Bus, AKS, Key Vault, and multiple identity flows is intermittently failing under load.

Architectural approach

Map the system first

Draw the end-to-end flow: client → front door/WAF → app/API → messaging → data → monitoring. Identify all dependencies (DNS, identity, network, config).

Define the failure

Define the failure: Is it latency, errors, timeouts, throttling, auth failures? Which layer? Which region?

Use observability as your compass

Correlate logs from App Insights, Log Analytics, and platform metrics. Look for patterns: spikes, throttles, 429s, 401s, DNS failures, connection resets.

Check the usual suspects

- Throttling (Event Hubs, Service Bus, Key Vault)
- Connection limits (SNAT exhaustion)
- Misconfigured retries / exponential backoff
- Identity token lifetimes / mis-scoped permissions

Architectural fix mindset

- Circuit breakers
- Retry policies
- Caching
- Proper partitioning / scaling
- Clear separation of sync vs async paths

2
Troubleshooting a multi-layer issue (networking, identity, app)

Scenario pattern

Example: A service in one subnet can't reliably call an API in another subscription using managed identity. Sometimes it works, sometimes it fails.

Architectural troubleshooting steps.

Layer 1 – Network reachability

- Verify DNS resolution (private endpoints, custom DNS).
- Check NSGs, UDRs, firewalls—are ports and directions correct?
- Use tools like Network Watcher, connection troubleshoot, and simple curl tests from a test VM/container.

Layer 2 – Identity & authorization

- Confirm the managed identity has the right role assignments on the target resource.
- Check token audience (resource) and scopes.
- Look at AAD sign-in logs and resource logs for 401/403 patterns.

Layer 3 – Application behavior

- Check connection pooling, retry logic, and timeouts.
- Ensure the app isn't caching stale tokens or mis-handling transient failures.
- Look at dependency telemetry in App Insights.

Layer 4 – Configuration & environment drift

- Compare working vs non-working environments: policies, NSGs, RBAC, feature flags.
- Look for "almost the same but not quite" differences—those are often the root cause.

Architectural takeaway

- From the fix, derive a pattern:
  - Standardized network patterns (hub-spoke, shared DNS)
  - Standard identity patterns (managed identity, role templates)
  - Standard app patterns (retry, timeout, telemetry)
- Document it as a reference architecture so the same class of issue doesn't repeat.