What Does A Good Design Look Like?

Some DOs and DON’Ts of technical design

Agenda

- Introduction
- Engineering Doesn’t Require Complexity
- Engineering Requires Understanding
- Design Decisions and Examples
- Summary and Review
- Questions / Insults / Solicitations

Introduction

- This talk is not “*how* to produce a good design” – “how” is specific to the customer’s technical requirements and what you are able to deliver/support

- This is “technically agnostic guidelines any generic design should follow”

- Why? – I’ve designed, implemented and supported many solutions over the years and it’s often too late when you discover a problem. This talk highlights how one can improve a solution design whilst still in the design phase (before it’s too late!) based on my own stressful experiances

Introduction

- All of the examples in this talk are real experiences from networks and projects I have worked on

- Sadly, I have many more examples and I continue to see the same issues

- The single biggest problem I continually see is technical overcomplication

Engineering Doesn’t Require Complexity

Engineering Doesn’t Require Complexity

- “Engineering” is typically associated with “technical complexity” – this is *the* biggest issue with engineering and design work

- Many respected figures agree:
  - “Simplicity is the ultimate sophistication.” – Leonardo da Vinci
  - “Simplicity is a prerequisite for reliability.” – Edsger Dijkstra
  - “$E=MC^2$” - Anon

- KISS

Engineering Doesn’t Require Complexity

- The technical aspects of your job are rarely the most demanding.
  It’s tough working in teams with:
    - mixed skill sets
    - mixed technical abilities
    - mixed availabilities
    - mixed [communicative] language proficiencies

- Techies don’t need to memorise $really_complex_thing

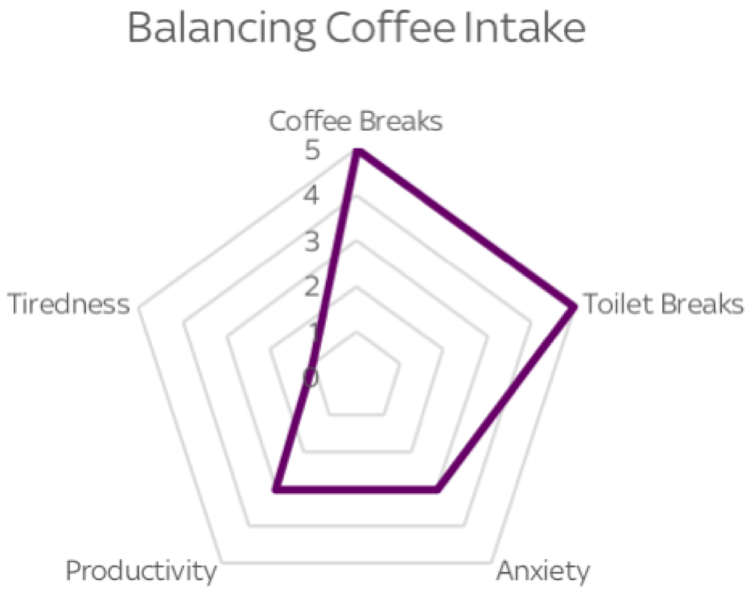
Engineering Doesn’t Require Complexity

- Engineers/Architects/Designers/Technicians need to be multifaceted and pragmatic. The E/A/D/T job is to translate between business requirements and **reasonable** technical methods

Engineering Requires Understanding

Engineering Requires Understanding

No solution design can implement, maintain and support itself. Every solution creates strain on different business resources, engineers need to balance the impact of their solution across the BUs.



Engineering Requires Understanding

2^{10}BC, ancient Chinese networkers followed the proverb:
如果您正在閱讀本文，那麼您正在使用Google翻譯！

Which roughly translates to:
“Good” designs are formulated by compiling the results from a collection of decisions. The result of each decision is the most balanced option between the impacted BUs of that decision.

Engineering Requires Understanding

For example; “which vendor should we use for project X?”

- One has to balance each of:
    - Cost (to please finance)
    - Lead time (to please project management)
    - Vendor SLAs (to please account managers)
    - Complexity (to please support teams)
    - Functionality (to please customers)
    - Compliance (to please auditors)
    - Standardisation (to please implementation teams)

Design Decisions and Examples

Design Decisions: Requirements and Cost

- DO: Design a network implementation that satisfies the customer/business technical requirements
- DON'T: Design what you think would be like, so cool, yeah, like OMG!
- DO: Keep in mind the budgetary constraints
- DON'T: Search for the cheapest possible solution

Example Scenario: Requirements

“I need 15Gbps of connectivity from A to B”

- Can easily add more 10Gbps links
- 10Gbps already “known” to operations

- A 100Gbps link would have 85% wasted capacity
- Do we have 100G tester and optics?

Example Scenario: Cost

"I need 15Gbps of connectivity from A to B"

- 10Gbps port and optics are cheap
- 10Gbps rental is cheap

- 100Gbps port and optics aren't cheap
- 100G rental is more expensive

Design Decisions: Scope and Deliverables

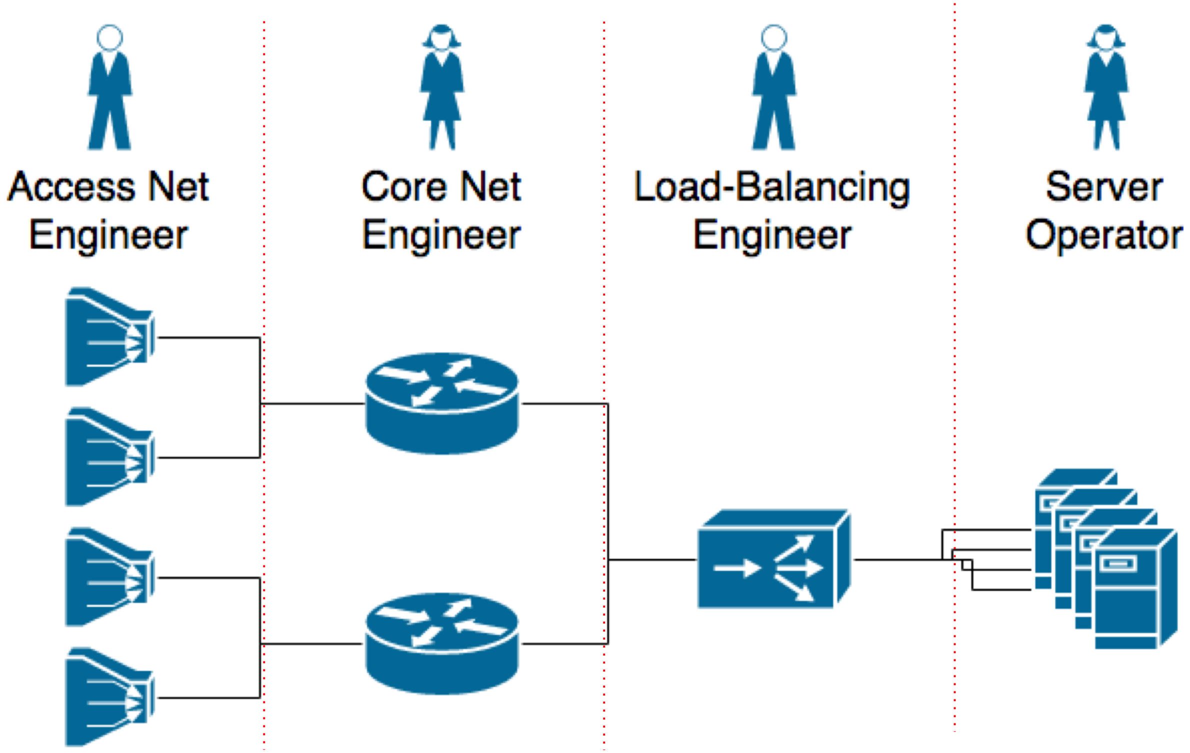
- DO: Clarify in as much detail as possible the project requirements/deliverables (ambiguity always leads to problems)

- DO: Confirm if the requirements can be broken down/aggregated up into smaller/larger deployment phases

- DON'T: Accept additions to the project scope or a reduction in the project deadline without explaining the impact and having it accepted

Example Scenario: Scope

In this example engineers designed and tested each section of the network in isolation; the final end-to-end test was a failure, very close to the project deadline.



Example Scenario: Deliverables

“Deliver an Internet connection at location X”

- Too specific:
  - “We’ll provide connectivity using four twisted pair copper cables, with each pair signaling at a frequency of 125 Mhz using a 5-level encoding scheme, to achieve a Layer 1 bit rate of 1.25Gbps, with a 2.5 volt peak average differential per twisted copper pair to maintain DC balance…”
- Not specific enough:
  - “A 1Gbps handover interface”
- Seems OK:
  - “A 1000Base-T Ethernet handover interface using RJ45 terminated Cat5e cable”

Design Decisions: Documentation and Support

- DO: Think about how you will document the solution. If you can’t easily explain it, how will others understand it?

- DO: Think about how others will have to troubleshoot the solution at 3AM (HLA, HLD, LLD, config templates, wiki/KB articles, cheat-sheets)

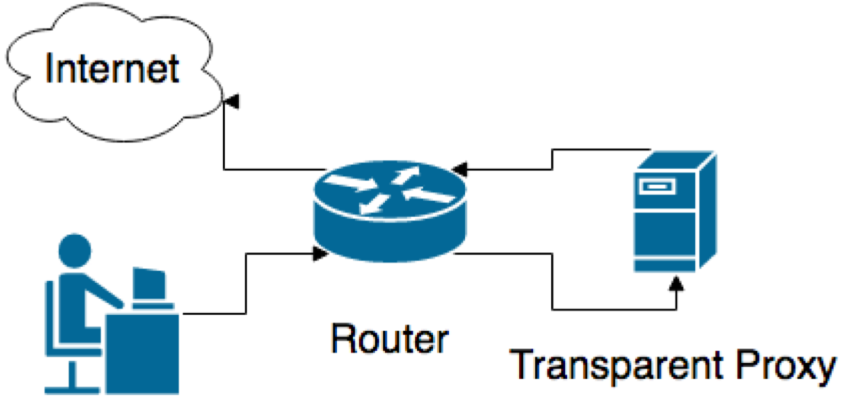
Design Decisions: Documentation and Support

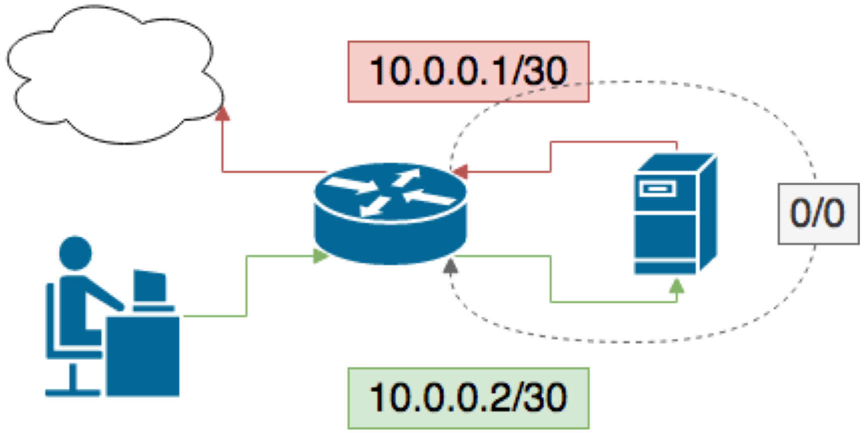
- DO: Try to be *so* specific in your documentation that you don’t need configuration examples (I hate config snippets!)

- DON’T: Mix disciplines, try to make failure domains that a single person or team can troubleshoot

Example Scenario: Documentation and Support

In this example, 1st line *network* support personal were expected to log into Linux *servers* and troubleshoot interface connectivity to ascertain why BGP on a router wasn't working!





Design Decisions: Standardisation and Monitoring

- DO: Standardisation is a major factor of scalability and reducing time to repair, use standardised products and services as much as possible

- DO: Accept non-standard *ideas*, every product catalogue starts empty

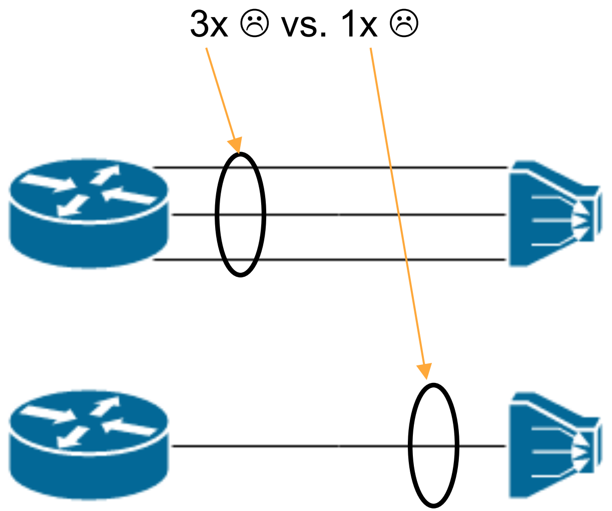
Design Decisions: Standardisation and Monitoring

- DON’T: Deploy what you can’t support. E.g. if your NMS can’t monitor a service, how will you provide service assurance?

- DON’T: Deploy anything that will increase your technical debt. No matter how simple a new technique maybe, who (else) can deploy/support/upgrade it?

Example Scenario: Standardisation

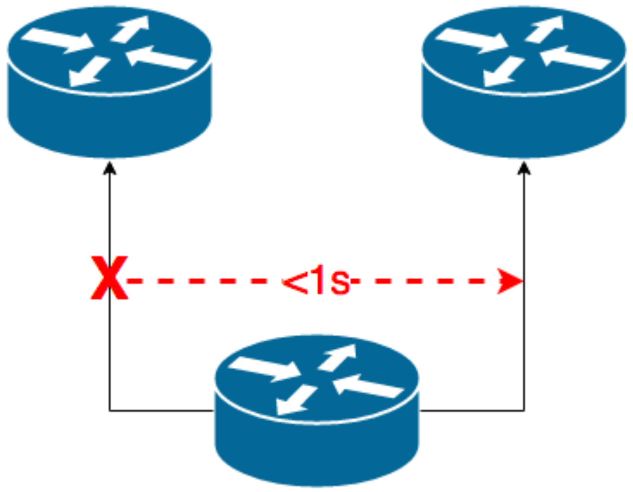
After foolishly accepting to productise bonded ADSL because, “it’s basically the same as regular ADSL” according to Sales and Marketing, I became the single point of contact for all bonded ADSL queries, provisions, support and escalations.



Example Scenario: Monitoring

A customer contract listed sub-second network failure detection and mitigation as a requirement for a standard service. It also required that the NMS be able to prove that the failure was detected and mitigated in less than 1 second.

Who polls once per second or faster?
How else could this be monitored?



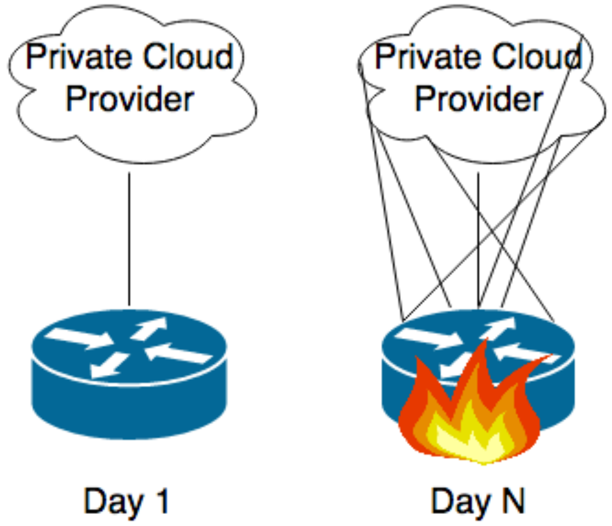
Design Decisions: Upgrades and Failures

- DO: Consider the upgrade path of the design for the reasonable future (12 to 24 months, nobody knows what will happen in 5 years time)
- DO: Consider the different failure scenarios that can happen and their individual likelihood, is there a dependency tree here with cascading failures?
- DON’T: Become distracted with every single failure scenario, focus on the requirements

Example Scenario: Upgrades

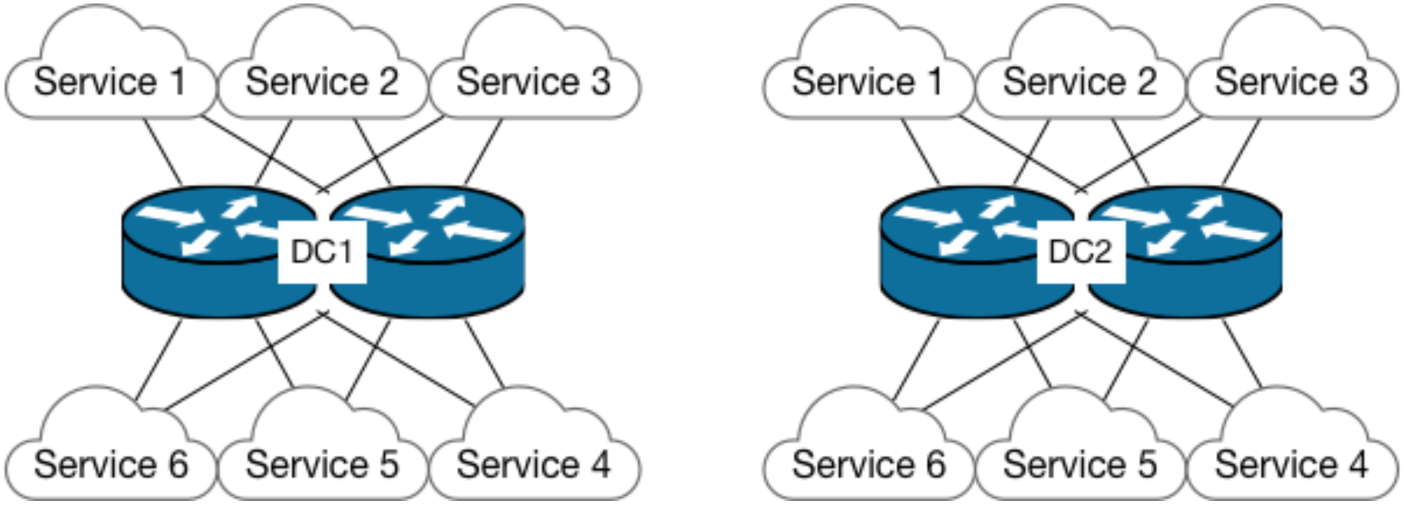
A customer required a one-off connection to a private cloud provider at a spoke/spur PoP, then word spread; a 2nd customer requested connectivity to the same cloud provide, then a 3rd, then a 4th, and so on…

That cloud provider *only* supported 10Gbps links and didn't support LAGs. This cost us ports and capacity in a PoP where we had little of both and overloaded our PE.



Example Scenario: Failures

We dual homed all services to two routers in DC 1, and replicated all services in DC 2 dual-homed to another pair of routers, N+N resiliency, WTFCGW?



Design Decisions: In-Life and Decommissioning

- DO: Think about how you will keep the design clean over time, will it "deteriorate" over time and become unclean?

- DON'T: Assume this 3 year deployment will still be needed next year, or that won't be here 15 years later

Design Decisions: In-Life and Decommissioning

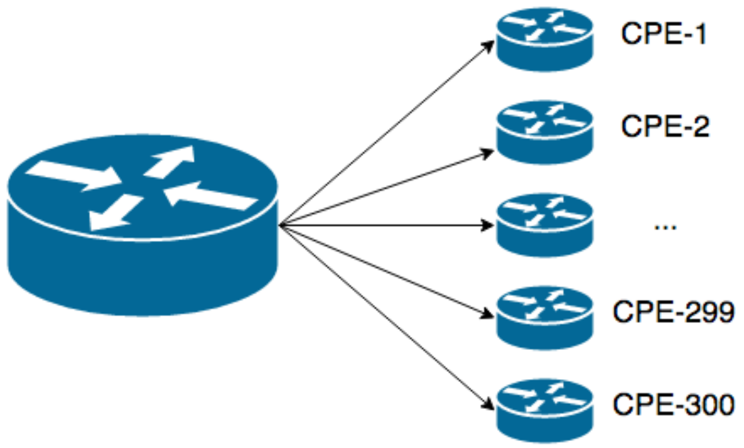
- DO: Think about how it will be decommissioned or partially decommissioned, don’t focus on provisioning only

- DON’T: Let the documentation “rot” or in-house knowledge become stale, have “refresher-sessions”

Example Scenario: In-Life

I once had to play a real-life game of ‘would you rather…’ and choose between adding 900 static routes or 300 BGP sessions to a satellite PE, which was already running the maximum number of vendor supported BGP sessions.

How can we be sure such a large number of routes/sessions are still valid 1/2/3 years from now?



In Summary…

Summary and Review

- Requirements: Define the requirement and solution as clearly as possible, demand clarification where ambiguity exists. Continuously refer back to the requirements and evidence fulfilment in your design documents.

- Cost: Keep in mind your budgetary constraints but don’t use sub-par materials to please finance.

Summary and Review

- Scope: Ensure the scope of the design is clear, explicitly state what isn't included (rather than implicitly by ambiguously not mentioning something). When it's agreed that something is out of scope or not required, record who approved that exclusion and why.

- Deliverables: Ensure everyone knows who's responsible for which areas of the design, and when each milestone is due.

Summary and Review

- Documentation: Document how something should behave, how it behaved when tested, what happened when testing failure scenarios, what happened during failures in production, are there any unknowns?

- Support: Break the design into smaller managed sections. Create cheat-sheets for troubleshooting these sections. Have operational handover and training sessions to educate the NOC. Have another one 12 months from now when everyone has forgotten. If a big outage occurs after 6 months, move back that 12 month review by 6 months.

Summary and Review

- Standardisation: This is a top priority with simplicity. Create standard products (config templates, monitoring templates, support templates) and reuse them throughout your designs. Can you easily hire someone to continue this work? Technical debt doesn’t only exist in a team in the present, but also in the future.

- Monitoring: If you can’t easily monitor it, how difficult will it be to add that functionality to your NMS? Will an upgrade of the NMS break that feature? Monitoring is not exempt from the simplicity/standardisation/supportability requirements, as soon as you can’t monitor a service you’re in trouble.

Summary and Review

- Upgrades: Try to think either a horizontal upgrade path (can we deploy more pizza boxes or add more line cards?) or a vertical upgrade path (what is the next generation of devices that will supersede the current ones?).

- Failures: They definitely will happen. Test the mostly likely ones to know what they look like on the CLI/via Syslog/NMS/from the customers perspective. What seemingly non-related infrastructure failures could impact this design?

Summary and Review

- In-Life Maintenance: In the best case scenario that the product/service is widely deployed, it shouldn’t be cumbersome to maintain with scale

- Decommissioning: If the documentation is up to date and all the components are standardised it *should* be simple but, the reality is config-rot or CMDB-rot

Summary and Review

- Complexity: Avoid complexity as much as possible, there is a direct correlation between complexity and support/billing/customer overhead

- Operations: When trying to balance between design decisions, default to what's best for your operations, not the customer; there’ll be other customers and you need to sleep at night

Questions?