What Does A Good Design Look Like?

Some DOs and DON’Ts of technical design

James Bensley @ UKNOF45

Introduction

- Who:
  - James Bensley
  - Designing/building/supporting/<u>decommissioning</u> for 10+ years
  - I’ve broken many things, and designed many monstrosities…

Introduction

- What:
  - Not this: “a good L3 VPN design must include …”
    - This is specific to *your* project

  - This: “technically agnostic guidelines any design should follow”

Introduction

- Why:
  - Experience has taught me the importance of *identifying the correct decisions* that need to be made early on, in order to deliver the required solution
  - Failing to meet expectations is worse than a “broken” solution
  - These problems aren’t going away (p.s. automation sucks!)

Introduction

- How:
  - Improve the solution design whilst still in the design phase

  - Share real experiences from networks/projects I’ve worked on (there are no “courses” on network design)

Engineering Doesn’t Require Complexity

Engineering Doesn’t Require Complexity

- “Engineering” is typically associated with technical complexity
  - Overengineering is my biggest issue with technical design work

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

- KISS

Engineering Doesn’t Require Complexity

- The technical aspects of your job are rarely the most demanding

  E.g., it’s tough working in teams with:
  - mixed technical abilities
  - mixed skill sets
  - mixed availabilities
  - mixed [communicative] language proficiencies

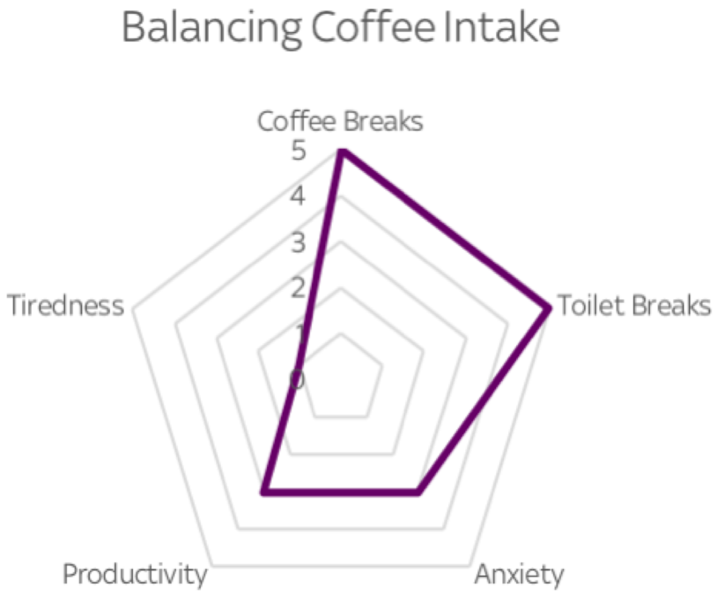
Engineering Doesn’t Require Complexity

- Techies don’t need to memorise $really_complex_thing
- Engineers/Architects/Designers/Technicians need to be multifaceted and pragmatic. The E/A/D/T job is to translate between business requirements and <u>**reasonable**</u> technical methods

Engineering Requires Balance

Engineering Requires Balance

- Every solution creates strain on different business resources, engineers need to balance the impact of their solution:



Engineering Requires Balance

- For example; “which device should we use for this project?”

  One has to balance the tradeoffs between each of the following:
  - Cost (to please finance managers)
  - Lead time (to please project managers)
  - 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 solution that satisfies the problem definition
- DON’T: Design what you think would be “like, so cool yeah!”
- DO: Keep in mind the budgetary constraints
- DON’T: Search default to the cheapest possible solution

Example Scenario: Requirements

Task: “I need 15G of connectivity from A to B”

- Can easily add more 10G links
- LAG/ECMP are “known” evils

- 100G link = 85% spare capacity
- Operationally simpler

Example Scenario: Cost

Task: “I need 15G of connectivity from A to B”

- 10G ports and optics are cheap
- 10G rental is cheap

- 100G port and optic is cheap*ish*
- 100G rental is not *as cheap*

Design Decisions: Scope and Deliverables

- DO: Clarify *unambiguously* what the project requirements are (ambiguity == problems)

- DO: Confirm if the requirements can be broken down (time > features)

Example Scenario: Deliverables

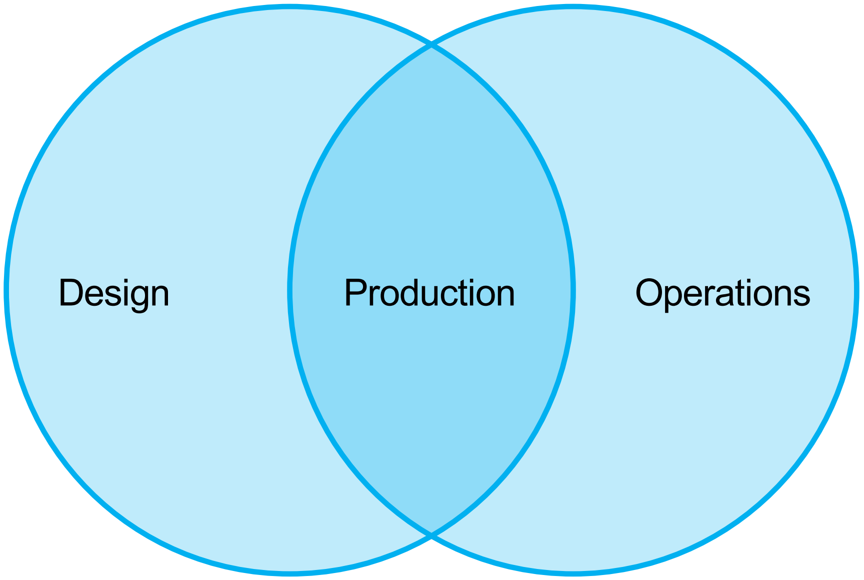
- Deliverable “provide an Internet connection at location xyz”
  - Too specific: “We’ll provide connectivity using four twisted pair copper cables, with each pair signalling at a frequency of 125 Mhz using a 5-level encoding scheme, to achieve a Layer 1 bit rate of 1.25Gbps, with a 2.5 volt peak average 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 on-call engineers will have to troubleshoot the solution at 03.00 AM
(HLA, HLD, LLD, config templates, wiki, KB articles, cheat-sheets)

Design Decisions: Documentation and Support



Design Decisions: Documentation and Support

- DO: Try to be *so* specific in your documentation that you don’t need configuration examples
  (they are the code comments of network engineering)

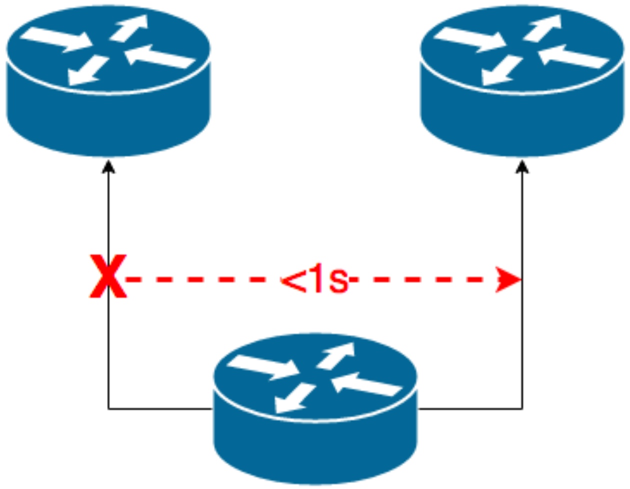
- DON’T: Mix disciplines, try to make failure domains that a single person or team can troubleshoot

Example Scenario: Monitoring

A customer RFP 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: Think about how you will keep the design clean over time, will it “deteriorate” over time and become unclean?
- DO: Consider the upgrade path of the design for the reasonable future

OSI Model	Potential Lifetime
Layer 4 - Transport	10s of months?
Layer 3 - Network	Single digit years?
Layer 2 - Data	Up to a decade?
Layer 1 - Physical	Multiple decades?

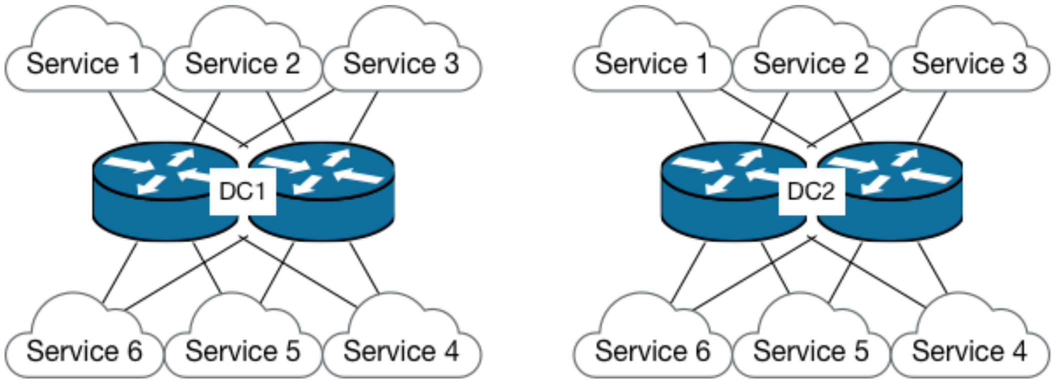
Design Decisions: Upgrades and Failures

- DO: Consider the different failure scenarios that can happen and their individual likelihood, is there a dependency tree of cascading failures?

- DON’T: Limit your consideration to technical failures:
  “if you design it they will *use* it”

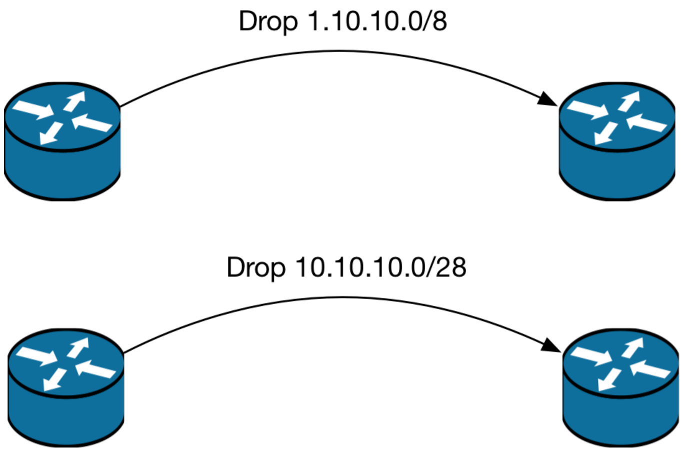
Example Scenario: Expected Failures

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



Example Scenario: Unexpected Failures

RTBH Routing → Prefix length validation?
RTBH Routing → Prefix validation?
RTBH Routing → Prefix purpose?



Questions?

send delayed thoughts of love/hate/confusion to: jwbensley@gmail.com

Review…

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.

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.

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 managable 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 another 6 months.

Review

- Operational Acceptance: Have someone else validate the deployment is working as expected. If they couldn’t it does work or they don’t understand the design. Share knowledge across team members/colleagues and other teams. Pro-actively train operational support teams rather than assuming they’ll get around to reading the documentation before there’s an outage.

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.

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?

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.

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.