David Swearingen - Data Society Questions
April 2 2020

Question 1

Assumption: I am not designing & building this in isolation, but I have a small team.

Key components of the plan:

- General matters, in no particular order of importance:
    - Identify the team resources and their skill levels. Some resources may be part-time. Sketch out the key tech areas you anticipate (e.g. front end, data tier, etc.) -- do you have the right skills mix? Start filling any gaps. (Don't forget UI/UX and graphic design, sometimes these are part-time positions or freelance.) Revisit this after the architecture has been designed.
    - Who's the tech lead?
    - Have the team members worked together before? What will the roles be? Do we have to hire or outsource for any particular aspect of the system?
    - What aspects of 'agile' or other methodologies do we wish to use? Agile practices should be tailored to the particular project and the team. E.g. some teams love pair programming, some don't, make a decision.
    - Do we have project management tools in place like Jira?
    - Who is the ultimate product owner? (This isn't always obvious.) Do they have the time bandwidth to apply to this effort?
    - What's the budget and schedule? How can the system be phased, what are the minimal set of features expected for launch?
    - Are the requirements documented clearly?
    - Review the requirements with the product owner. This likely would involve several meetings. Assign some general phases and priorities to the requirements.
    - There should be some back and forth between the team and the product owner to refine priorities.
        - Example: a feature that the product owner originally said was "important" turns out to be expensive to develop. Go back to the product owner and "negotiate" with her to establish just how important that feature is: tell her you can do it, but several of her other features will have to wait. After hearing this, this clarifies her thinking and she reduces the importance she originally assigned to that feaature.
    - Make sure you're finding the balance between preparing for the future (product successful, tons of traffic) and building capability that may never get used (product not so successful, limited traffic). This requires good factoring. Example: you project that a moderately successful product will not require a message queue. Should you build in the message queue anyway, even though that drives up the cost? (this is just a fictional example, I'd probably build in a queue no matter what). The answer may be to factor the code carefully with clean abstractions and loose coupling, allowing us to insert a queue easily later on.

- Technical matters:
    - First of all, there are few truly "new" problems in building systems like this, so look for best practices and design patterns that have made other teams successful. This will require some reading, talking to tech experts inside and outside the firm, even if they won't be on the project. Likely there will be challenges that no one on the team has ever done.
    - Identify and prototype key pieces of the system that the team feels are new to them. Example: maybe someone on the team has used Kafka but it was three years ago and it was on-prem. Now the team agrees that we should use AWS. What's different about AWS's message queue software that could bite us in the rear if we aren't aware of it? Test this.
    - Create a working end-to-end system quickly as a prototype to identify problems. (This is called 'tracer bullets' in one of the canonical software engineering books.) This would be a system with a ugly front-end and lacking any refinement, but it shows end to end functionality and this could surface key problems that no one thought of.
    - Tech stack:
        - Let's assume this is cloud e.g. AWS/Google/Azure for the sake of simplicity. *(IMPORTANT NOTE: some customers like federal defense/intel can't do any cloud, for security or other reasons. Some have cloud but it's unique and you have to know what its limitations are.)* Make sure the team is comfortable with cloud (single point of failure). Do some back of napkin pricing calculations and inform your product owner, cloud costs can be high, don't forget I/O costs!
        - S3 storage (or equiv)
        - Middle tier: Node.js could be a good choice, depends upon team comfort level and requirements. Other options are python, java.
        - Decide upon traditional EC2 hosting, or numerous options like Lambda, Serverless.com. This is a pretty big decision, needs to be discussed and researched.
        - Message queue, Kafka probably has the best mindshare and reputation
        - Front-end: likely React/Angular/Vue provide the best abstractions
        - Database: lots of options. Gotta choose between relational and NoSQL. The latter is schema-flexible, but can have gotchas.
        - Don't forget a place to log events. Something like Papertrail or an equivalent
        - PagerDuty or other systems for messaging ops team if there's a problem, and then managing the remediation process
        - Authentication: how will this be done? SAML? In the cloud? Do you have an expert to figure this out?
        - External service dependencies? Payment systems, document signing, etc., what are they, are they reliable and affordable and will they be around for years?
        - Security audit: is this type of system a good target for hackers? Even if it's not, you still need to secure it. Who will you use to white hat the app and review your tech architecture early on to spot bad patterns? What about ongoing security reviews?
        - Project management and version control tools like JIRA, git
        - Profiling and performance tooling (depends upon language)

- CI / CD should be discussed, what approaches is the team going to take? This is a big subject.
- Will something like Google or Adobe Analytics be used? What are the implications of this?

Question 2

Key to success here is proper planning, rollback plans, backups, etc! *You can't solve this problem easily and effectively without having planned and practiced for it from the beginning.* This needs to be part of the requirements and these scenarios need to be documented and designed during the initial development!

But in terms of the actual process: having a message queue in place is likely part of the key to handling upgrading services since this buffers data while other systems are swapped out. So data can continue to flow in to the queues, however you still need front-end working full-time, this likely would be done by using a load balancer configured to switch over traffic to new back-end server instance.

I'm not an expert on this subject, but **I know what I don't know**, and I respect the potential complexity of this, which is critical. Experts should be engaged early in the project planning to address planning for this.

Question 3

First steps would be checking log files for various aspects of the system, this assumes there is good logging in place to begin with across the application. (Did we have a plan in place for handling this, e.g. is there a roster for who responds on this day?) The logging system (like Papertrail) would be full text searchable to facilitate this.

What is the exact nature of the failures, e.g. can't write to storage, or some third-party service we depend upon failed? Did we commit a recent change to the app and could that have caused the problem? Have we received any service outage reports from the cloud provider?

Question 4

I had a customer several years ago that needed a dashboard showing metrics on data for one of their lines of business. During the proposal phase I proposed and strongly encouraged the use of an off-the-shelf data warehouse. But the client product owner (who was not technical) insisted that they didn't want any kind of COTS or software that required a license, no matter the cost, because they

hadn't budgeted for that. This forced me to build essentially a data warehouse from scratch, which was quite complicated, and not ideal.

In hindsight, I would have found a way to persuade the customer that the best overall solution would have been to use an off-the-shelf data warehouse, or decline the job. So this was a lesson not relating to the technology per se, but rather to how to manage a customer, and when to say "no".

Question 5

Crucial traits for a successful software team:

- Trust and respect amongst team members
- Humility. Arrogance can be poisonous to a team
- People that like to share and aren't competitive with their team members
- Good communication practices (verbal and non-verbal e.g. commenting code and commits) within the team but also with others. This surfaces problems quickly.
- Diversity of experiences, both technical and non-technical
- Sense of humor
- Similar philosophies on the important tech and project matters (like the importance of tests)

There are others but these are some of the most important that come to mind.

Question 6

Code is in https://github.com/bethesdamd/data_society_quiz

- Note: print() statement was placed outside of output() in order to facilitate unit testing.
- I have omitted the amount of commenting I would normally do in a production system
- A unit test is included in the repo

Question 7

Code is in https://github.com/bethesdamd/data_society_quiz

Notes:

- Code requires 'money' python library. This resolves complexities around how to properly represent currency in python, allows other currencies, etc.
- Please **pip/conda install money**. I didn't do any packaging.
- **Run Question7_B_main.py**

Part B:

- In a "real" streaming system, likely I would implement this with a message queue (e.g. Kafka) streaming the individual objects to a Topic, and then the logic would be done in a stateless service like perhaps AWS Lambda.
- So my implementation is a simulation, there's a main program Question7_B_main.py that reads the input json file and then calls a processor which is in module Question7_B_module.py. This simulates decoupling. The process() function simulates writing running the running total value to a queue, which is actually the local file system, meant to mock a queue.
- So to run this, **run Question7_B_main.py**