**Advanced Software Engineering: Reflective Essay**

For the Advanced Software Engineering module we were asked to create a mobile app and server with a variety of features of increasing complexity. Initially the app was required to display a user’s GPS location on a map, and later this became the basis of a valuation tool for property developers based on Price Paid Data (PPD) for the UK housing market. To fulfill the brief we delivered the following components:

* Android App, featuring house valuation heatmaps
* iPhone App, featuring house valuation heatmaps
* Node.js bespoke MVC API
* Data-preprocessing script to load prepare available PPD for import to the API’s database

The project had a number of unique and common challenges, which we were required to navigate in order to complete the deliverables. Some of these challenges included:

* New project team, necessitating that we quickly built strong working relationships with new people.
* A blank-slate in terms of tooling and automation. This required a lot of infrastructure setup in order to work effectively, but also no legacy systems to wrestle with.
* Geographically distributed team with very different schedules.
* Competing commitments and unexpected or changing circumstances to navigate, such as commitments for other course modules, work, family, as well as illness and one team member leaving the course.
* Learning new technologies and adapting plans in order to answer unfolding requirements.

To handle these challenges and meet the deliverables we attempted to incorporate a range of Software Engineering ideas and methodologies. Here we discuss some of the ideas we have used and how they have played out during the project.

**Specification**

In defining a specification we attempted to both explore and clarify the customer’s requirements. By exploring the requirements with clear use cases we were able to quickly discover implicit requirements and unstated user expectations. For example, a heatmap on a geographical map carries a user expectation for the ability to move around and view the heatmap, and consequently, but potentially less obviously, a user expectation for the ability to return to user’s the original location.

We used three approaches to gathering requirements in communication with the customer, which were a specifications document, a set of low-fidelity wireframes, and an architecture diagram. These helped us join dots, quickly see gaps in our current understanding of the requirements, and importantly define our response to the initial team in a way that the customer could easily agree or disagree with, and that the team could clearly follow. The user story format we followed in the specification document gave us a good structure for summarising the requirements in an easily understandable way, but also defining precise testing requirements for the team. While the diagram & wireframe became more of a practical reference in reality, the user stories in the specification document gave us our Trello tasks, and we were able to track the progress of each user story with a kanban board.

**Flexible Architecture**

Without always having visibility of what the next task, or functionality will be in this project, it was important that the applications we built could embrace change. We attempted to allow for this defining discrete components of the overall system and by using MVC architecture in each component. This in theory meant that parts were replaceable, although in reality we did not encounter the need to.

We were able to use this approach to scale our map results to display results from an initially limited region of Brighton to the whole of England, and to scale our offering to include a second client application built for iOS. We could have potentially taken this further to include other future clients such as apps for Windows phones or a Progressive Web App given more time. API version urls would also have assisted with this, as the API could have continued to progress in features while still maintaining stable support for all clients that had not been updated. Version paths were a consideration, however the Node.js server we were using made it very difficult to run more than one version of the API simultaneously on the same port.

We also gained some flexibility by including reproducibility where possible, and most significantly in the data preprocessing script. By creating this script with the assumption that it would be re-run, it allowed us to potentially automate data processing if the source data updates (which it does annually). More directly relevant for this project was that this approach allowed us to quickly fix any issues with imported price paid data.

**Deployments, logs and error handling**

Two other features that gave us flexibility were the ability to deploy server updates quickly, and very robust logging and error handling for the API. These features allowed us to make changes easily, but also to quickly diagnose the impact of those changes and fix bugs that occurred from them.

We initially we put a lot of work in to setting up a deployment process based on AWS CodeDeploy and TravisCI. This had several issues, which meant that we ultimately abandoned this route in favour of a linux alias. One problem we had with CodeDeploy and TravisCI, was that each time TravisCI ran it took a big chuck out of our AWS free tier usage limits. This meant that TravisCI had the opposite effect of what we were trying to achieve with a CI tool, as developers were actually deploying less often, in order to lower AWS usage and remain in the free tier. Adding a bash alias increased deployments significantly and also meant developers were more likely to communicate with other developers prior to launching updated code. So in this instance with a distributed team, an automated but manually triggered deployment worked best for us.

Once a deployment has been made, our custom bash alias command immediately runs “tail -f” on the relevant log files, and then proceeds to run all of the tests against each API endpoint in turn. The granularity of our logging and error messages has meant that a developer would immediately see the results of their changes, as well as what aspects of their changes had caused specific tests to fail. If the developer had deployed a bug, they would often know within 30 seconds of the deployment, and have been provided with the information they needed to find and diagnose the problem. This in turn lead to confident deployments as deployed bugs could be quickly resolved, which encouraged developers to deploy frequently and reiterate often.

**Hosting investigation**

Before settling on AWS, we also ran some checks on EC2 and compared it against other providers, such as Azure, Heroku, Google Cloud Platform. Here is an example of some of the things we were looking for in this investigation:

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Notes** | **EC2** |
| Number of steps to get a basic HTTP server running | How easy the system will be to administer | 15 |
| $/month | What it will cost to keep the system running | £0 (free tier) |
| Power cycle time (seconds) | Minimise downtime between server upgrades/reboots | ~90 seconds |
| Time to connect (seconds) | Minimise effort/wait time during development | ~1 second |
| HTTP request latency | Performance important for best UX | 0.104 seconds |

The difference between major vendors for most key technical performance criteria was not noteworthy. Without an obvious difference in technical capabilities that were likely to affect us in this project, we made a decision based on which provider we thought it would be most valuable to learn. The team settled on Amazon AWS, which provided us with a lot of learning opportunities as well as a lot of opportunities to scale infrastructure.

**Testing**

As a team we had mixed success with testing. We created around 45 automated tests, however there were limitations to our approach and areas for improvement.

To test the system we used a combination of unit tests, automated UI tests, and an API testing tool. This gave us a good coverage and allowed us to perform unit and integration testing from a variety of angles. We tested for expected conditions, as well as unexpected conditions (null values, broken values, no data etc).

Despite this, our testing was let down by a limitation in our approach and an unfortunate error. The limitation was that due to IT issues we found it difficult to emulate each other's apps, which combined with our physical separation meant that we were unable to do much user acceptance testing. This was a real problem for us as this was something we had planned to do a lot of in the initial project plan. Ideally, we would have spent more time working together in group working spaces, however work schedules did not allow for this.

This situation, combined with a misunderstanding lead to an unfortunate error where a bug in one demo wasn't fixed in the next. In the first of the demos several bugs had been seen and spoken about concurrently, leading the team note, fix and test all but one of the bugs mentioned. User acceptance testing may have helped notice the slip.

**Team Collaboration**

The biggest hurdles to team collaboration were due to different physical locations and schedules. This meant that the team were only able to meet up occasionally for catch-ups, but were never able to work in the same location and were often working on the project at different times in the day. This is a step further than the commercial problem of working between offices in different locations and timezones. Colleagues in different timezones do at least have an agreed norm in terms of working schedules (i.e. 9-5), and a standardisation for the time offset (+5 hours for EST etc). In a university group project, schedules are sporadic and it is very difficult to know when other people are likely to be working on the project.

To counter this, we spent a lot of time in the early stages of the project setting up the right tools to make communication as smooth as possible. Tasks & issues were tracked through Trello with weekly meetings to confirm progress, we created specification documents and diagrams to align expectations, and very little happened without alerts and conversation on Slack. In total the team sent each other over 3,600 slack messages during the project.

The team was in constant communication at all stages of the project, and from this perspective collaboration worked well. However, nothing beats being in the same room. We spent a lot of time as a team updating each other on what were working on, but even then a slack message is only ever part of the picture. Ultimately this limitation created problems in our demonstrations as it was difficult to do good user acceptance testing on each other’s work. We discovered one bug in particular during the last demo which had its roots in a miscommunication around what the server was expected to return in one instance.

Another soft goal of the project was the opportunity to learn new technology. We believe this is an area we made the most of, as the team took the initiative to learn the following technologies:

* Android SDK
* Swift/iOS
* Node.js
* Amazon AWS

We were able to do this because on the whole the team were collaborating well due to our tooling and specifications. This meant that we weren’t in a situation where one or two people were completely “owning” the project, with other people tip-toeing around completing peripheral tasks. Each team member was able to make significant contributions to the project.

Initially we began the project with the intention of assigning complexity estimates to tasks, so that we could determine sprint selections. This was difficult to accomplish as in most cases the tasks we were building were on technologies that we were still learning, so it was difficult to account for the time needed to investigate how to carry out the task. There was also no accurate sprint budget to determine how many estimated tasks could be included in each sprint, as team members were working around other commitments and could not always give confident assessments of how much time they would have each week.

This lead to us to making sprint selections based on MoSCoW prioritisation and removing blockers, rather than sprint points/budget. Prioritisation without specific budget worked very well for us in this context. We typically completed our sprint selection each week as the clearly communicated priority of the selected tasks meant that the team “made time” for the high priority tasks when necessary, despite the ambiguity of available time.