Capstone Project

Software Architecture

Semester 1, 2024

Richard Thomas & Brae Webb

Summary

Throughout the software architecture course, you have learnt about a subset of quality attributes of concern to software architects. You have also been exposed to a number of techniques to satisfy these attributes. Now, as the capstone project, you are required to

- propose a non-trivial software project,
- identify the primary quality attributes which would enable success of the project,
- design an architecture suitable for the aims of the project,
- deploy the architecture, utilising any techniques you have learnt in or out of the course, and
- evaluate and report on the success of the software project.

The successful completion of the project will result in three deliverables, namely,

i a proposal of a software project, the proposal must clearly indicate and prioritise two or three quality attributes most important to the project's success,

ii the developed software, as both source code, and a deployed artifact, and

iii a report which evaluates the success of the developed software relative to the chosen quality attributes.

Your software deliverable must include all supporting software (e.g. test suites or utilities) that are developed to support the delivered software.

1 Introduction

We have looked at several core quality attributes in this course, and will continue to look at more over the remainder of the semester. These attributes were selected because they are key concerns of many real-world software projects. In this project, you will have an opportunity to explore some of the fun of industry. You will take the role of an entrepreneur, software architecture, developer, and operations team.

As entrepreneur, you have proposed a project idea. These proposals have been evaluated by your peers and the teaching team. You have been allocated to a project based on interest you have indicated by voting on proposals and on your previous coursework experience.

As a team you now need to perform the roles of software architect, developer, and operations team. You should design the basic structure of the initial software architecture, based on the scope, functionality, quality attributes, and evaluation plan from the proposal. The details of the architecture are expected to evolve as you start implementing parts of the system.

Part of the assessment will be how the architecture evolves in response to what you learn during development. You need to write Architectural Decision Records (ADRs) for each decision you make about the design of the architecture [1]. These are to be recorded in your GitHub repository so that the marker can see how your architecture evolved and the reasons for the decisions you made.

2 Software

You need to implement a software system that delivers a Minimum Viable Product (MVP)[1]. The MVP needs to implement a usable core of the system's functionality, which demonstrates that the architecture could deliver the full system functionality. The MVP also needs to allow the software architecture to be tested to determine if it can deliver the project's important quality attributes.

You may renegotiate the scope of the system during the project, if you determine that certain aspects of the original scope are not feasible within the project time constraints. The earlier you do this, the less it will impact on your final result. You will not explicitly lose marks for renegotiating scope, unless the revised scope limits your ability to adequately test important quality attributes. But, late changes to scope are likely to have a flow-on effect that could reduce the quality of your final deliverables. This means that you should attempt to implement some of the riskier parts of the project early.

3 Evaluation

You need to test the software system that you implement to demonstrate how well its architecture supports delivering system functionality and its quality attributes. This evaluation should be based on the proposal's evaluation plan, but should ***not*** be limited to only what is in that plan. You will be assessed on how well you test your system in terms of both functionality and quality attributes. Discovering issues with the system or its architecture during testing will not adversely affect your marks for the evaluation component of the assessment.

A section of your project report needs to summarise the test results and provide access to the full suite of tests. You should automate as much of the testing as possible. Any manual tests need to be documented so that they can be duplicated. The results of all manual tests need to be recorded in a test report. This may be a section of the project report, or it may be a separate document with a link to it from the project report. You need to include test code and test infrastructure in your project's repository.

4 Report

The report should include the following content.

Title Name of your software project.

Abstract Summarise the key points of your document.

Changes Describe and justify any changes made to the project from what was outlined in the proposal.

Architecture Options What architectural design patterns were considered and their pros and cons.

Architecture Describe the MVP's software architecture in detail.

Trade-Offs Describe and justify the trade-offs made in designing the architecture.

Critique Describe how well the architecture supports delivering the complete system.

Evaluation Summarise testing results and justify how well the software achieves its quality attributes.

Reflection Lessons learnt and what you would do differently.

[1] https://www.agilealliance.org/glossary/mvp/

You do not need to have sections for each topic above, though your report needs to contain the content summarised above. For example, the description of the architecture could include discussion of trade-offs. Similarly, the critique and evaluation could be combined so that both are discussed in relation to an architecturally significant requirement (ASR) [2].

When writing your report, you may assume that the reader is familiar with the project proposal. You will need to describe any changes your team has made to the original proposal. A rationale should be provided for each change. Small changes only need a brief summary of the reason for the change. Significant changes to functionality of the MVP, or changes to important quality attributes, need a more detailed justification for the change. You should provide a reference and link to the original proposal.

Compare and contrast different architectural design patterns that could be used to deliver the system. Explain the pros and cons of each architectural design pattern in the context of the system's functionality and ASRs. Justify your choice of the architectural design pattern you used in your design.

Describe the full architecture of your MVP in enough detail to give the reader a complete understanding of the architecture's design. Use appropriate views, diagrams and commentary to describe the software architecture. You should describe parts of the detailed design that demonstrate how the architecture supports delivering key quality attributes [2]. (e.g. If interoperability was a key quality attribute, you would need to describe the parts of the detailed design that support this. For example, how you use the adapter design pattern to communicate with external services.)

Describe any trade-offs made during the design of the architecture. Explain what were the competing issues[2] and explain why you made the decisions that resulted in your submitted design.

When describing the architecture and trade-offs, you should summarise and/or reference ADRs that relate to important decisions that affected your architecture.

Your critique should discuss how well the architecture is suited to delivering the full system functionality and quality attributes. Use test results to support your claims, where this can be shown through testing. For quality attributes that cannot be easily tested (e.g. extensibility, interoperability, ...), you will need to provide an argument, based on your architectural design, about how the design supports or enables the attribute. Some quality attributes (e.g. scalability) may require both test results and argumentation to demonstrate how well the attributed is delivered.

Summarise test plans and test results in the report. Provide links to any test plans, scripts or code in your repository. Where feasible, tests should be automated. Describe how to run the tests. Ideally, you should use GitHub Actions[3] to run tests and potentially deploy artefacts.

Your report should end with a reflection that summarises what you have learnt from designing and implementing this project. It should include descriptions of what you would do differently, after the experience of implementing the project. Describe potential benefits or improvements that may be delivered by applying the lessons you have learnt during the project. You will not lose marks for identifying problems with your architecture or software design.

5 Repository

Your team will be provisioned with a repository on GitHub. All source code, documentation and support artefacts are to be committed to the repository.

- Model artefacts (e.g. Structurizr DSL or PUML files) should be stored in the `/model` directory.
- ADRs are to be stored in the `/model/adrs` directory.
- The report must be stored in the `/report` directory.
- The link to your demonstration video must be in a file called `demo.md`, in the root directory of your repository.

[2] "Forces" in design patterns terminology.

[3] https://docs.github.com/en/actions

Do ***not*** commit large binary files to the repository. (i.e. Do not commit Word documents or frequently changing PDF files to the repository. Do not store your demonstration video in your repository.) It is recommended that you use LaTeX, or possibly markdown, to write your report. If you use LaTeX, you should use GitHub actions to produce a PDF of the report.

Your final submission will be what is in the main branch of your repository at the due date of 15:00 on 3 June 2024.

6 Academic Integrity

As this is a higher-level course, you are expected to be familiar with the importance of academic integrity in general, and the details of UQ's rules. If you need a reminder, review the Academic Integrity Modules[4]. Submissions will be checked to ensure that the work submitted is not plagiarised.

All code that you submit must be your own work or must be appropriately cited. If you find ideas, code fragments, or libraries from external sources (e.g. Stack Overflow), you must cite and reference[5] these sources. Use the IEEE referencing style[6] for citations and references. Citations should be included in a comment at the location where the idea is used in your code. All references for citations must be included in a file called `refs.md`. This file should be in the root directory of your repository.

You are encouraged to use a generative AI tool (e.g. copilot) to help you write the source code for this project. The expectation is that the software architecture and detailed design are your team's own work. Create a file in the root directory of your repository called `ai.md`. In this file, describe how you used generative AI to develop your project. (e.g. We prompted ChatGPT with a description of our design and refined the provided code via further prompting and manually revising the code.) Provide examples of prompts provided to any generative AI tool.

In `ai.md` indicate which files contain code produced with the assistance of an AI tool. Estimate how much of the code was produced by the tool and how much was your own work.
(e.g. `logic.py  40% generated`)

You may use libraries to help implement your project. The library's license must allow you to use it in the context of your project. All libraries used in your project must be listed in a file called `libs.md`. You must include a link to each library's homepage. This `libs.md` file must be in your repository's root directory.

Uncited, unreferenced or unacknowledged material will be treated as not being your own work. Significant amounts of cited material from other sources will be considered to be of no academic merit. Having an AI tool produce significant amounts of source code is acceptable, *if* the design is your own and you have verified that the code is correct.

7 Demonstration

Your team needs to demonstrate your project's functionality and how well it achieves its goals. This should include demonstrating how quality attributes are achieved, or briefly summarising how the architecture facilitates delivering a quality attribute.

Your project demonstration will be a video. Provide a link to the video in a file called `demo.md`, stored in the root directory of your repository. Do ***not*** store the video in your GitHub repository. The link may be to the video on a platform like YouTube, or a file sharing site from where the video may be downloaded. If you upload the video to a platform like YouTube, you may make it private. If it is a private video, you must share it with `richard.thomas@uq.edu.au`, `evan.hughes@uq.edu.au`, `mdarafat.hossain@uq.edu.au`, `zaidul.alam@uq.edu.au`, `r.wibawa@uq.edu.au`, and all of your team members. The video must remain

[4] https://web.library.uq.edu.au/library-services/it/learnuq-blackboard-help/academic-integrity-modules

[5] https://guides.library.uq.edu.au/referencing

[6] https://libraryguides.vu.edu.au/ieeereferencing/gettingstarted

available until *at least* 31 July 2024. Viewers must be able to easily see what is being demonstrated and read any text or images. Audio must be clear and comprehensible.

The video should be approximately as follows.

3 min Introduction to the project.

3 min Demonstration of the functional requirements.

3 min Demonstration of the non-functional requirements.

3 min Overview of the software architecture.

3 min Summary of your reflection on lessons learnt from implementing the software.

The total duration of your video should be ***less*** than 15 minutes.

Introduction Briefly introduce the project context and summarise the delivered functional and non-functional requirements. Mention any differences between what was originally proposed, what was renegotiated, and what was delivered. Briefly explain why changes in scope were made. The person who may have approved a change in scope may not be the person marking your demonstration. If you did not deliver everything in the revised scope of the project, the marker needs to know why that occurred.

Functional Requirements Demonstrate the key features of the software. You do not have time to demonstrate every feature of the software. Plan your time wisely to highlight the completeness and quality of your delivered system.

Non-Functional Requirements Show how well the software delivers its important quality attributes. This may take some thought and planning to demonstrate within a short time frame. Delivery of some non-functional requirements can be shown by test results. Delivery of other non-functional requirements may be shown through a combination of tests, metrics, and commentary.

For example, you cannot show ten minutes of k6 testing to demonstrate scalability. You could provide screenshots of different stages of the testing, or an edited video of parts of the testing. You would provide commentary summarising how the testing was done and explaining how well the system scaled under different loads.

For security, you could show results of simple fuzz testing of APIs. You could then show examples of parts of your design, explaining how it demonstrates following key security design principles.

For extensibility or interoperability, you could calculate one or more complexity metrics for parts of the design. You could then use the data from these metrics to support an argument as to why the design was extensible or had a simple interface. For example, if many interfaces could be shown to have high cohesion and there was low coupling between different modules or services in the design, you could argue how this shows that the design is likely to be extensible. You could measure documentation for interfaces or APIs, and use that to argue that mechanisms used to extend the design, or that the APIs, were comprehensible.

These are examples to help you to start thinking about demonstrating how your design delivers non-functional requirements. They are not a definitive list of the only or best approaches. For the demonstration, focus on the most important non-functional requirements for your project. You should discuss your ideas with course staff if you are unsure of the effectiveness of an approach.

Software Architecture Provide an overview the system's architecture. Briefly explain how well it supports delivery of the MVP's, *and* the full system's, functional and non-functional requirements.

Reflection Summarise the lessons you learnt from implementing the software. What would you do differently and why? Explain how you would apply those lessons to design a different architecture or take a different approach to implementing the project. Or, explain how the lessons learnt demonstrate that you made good choices at each stage of development.

Presentation There are no constraints on who in your team presents in the video. One person could present all parts of the video, or you could have different people presenting each part. Assume that the viewer has read the project proposal but may not yet have read the project report.

8 Grading Criteria

20% Extent to which project's scope was delivered.

15% Suitability of architecture to deliver system goals.

20% Quality and thoroughness of testing.

25% Clarity, accuracy and completeness of architecture's description.

20% Insightfulness of architecture's evaluation.

Criteria	Standard						
	Exceptional (7)	**Advanced (6)**	**Proficient (5)**	**Functional (4)**	**Developing (3)**	**Little Evidence (2)**	**No Evidence (1)**
System Scope 20%	MVP's originally proposed functional & non-functional requirements, or those agreed & documented early in the project, are fully delivered.	MVP's originally proposed functional & non-functional requirements, or those agreed & documented early in the project, are delivered with small variances.	MVP's functional & non-functional requirements were revised & documented later in the project, and are almost fully delivered.	All important functional & non-functional requirements are delivered but some other requirements are not, whether or not original plan was revised.	Most important functional & non-functional requirements are delivered, whether or not original plan was revised.	Some important functional & non-functional requirements are delivered, whether or not original plan was revised.	Few important functional & non-functional requirements are delivered, whether or not original plan was revised.
Architecture Suitability 15%	Delivered architecture, supplemented by the design reflection, is very well suited to delivering all specified functional & non-functional requirements, including an appropriate level of security.	Delivered architecture, supplemented by the design reflection, is well suited to delivering almost all specified functional & non-functional requirements, including an appropriate level of security.	Delivered architecture, supplemented by the design reflection, is fairly well suited to delivering the key functional & non-functional requirements, including a mostly appropriate level of security.	Delivered architecture, supplemented by the design reflection, is capable of delivering most key functional & non-functional requirements, including a mostly appropriate level of security.	Delivered architecture, supplemented by the design reflection, requires workarounds in a few cases to deliver key functional & non-functional requirements. Design has one or two obvious security issues.	Delivered architecture, supplemented by the design reflection, requires workarounds in several cases to deliver key functional & non-functional requirements. Design has a few obvious security issues.	Delivered architecture, supplemented by the design reflection, makes it difficult to deliver many functional & non-functional requirements. Design does not appear to consider security issues.
Testing Quality 20%	All functional & non-functional requirements, & architectural components are well tested (or are described well in a test plan) and, where feasible, are automated.	Most key functional & non-functional requirements, & key architectural components are well tested (or are described adequately in a test plan) and, where feasible, are mostly automated.	Most key functional & non-functional requirements, & key architectural components are fairly well tested (or are described fairly adequately in a test plan) and, where feasible, many are automated.	Most key functional & non-functional requirements, & key architectural components are fairly well tested (or are described fairly adequately in a test plan) and, with some attempt at automation.	Main test cases for most key functional & non-functional requirements, & key architectural components are fairly well tested (or have some informative description in a test plan).	Main test cases for a few key functional & non-functional requirements, & key architectural components are moderately well tested (or have a general description in a test plan).	Testing is poor, superficial or extremely limited. Or, extent of testing cannot be determined from submitted artefacts.
Architecture Description 25%	Clear, accurate, concise & complete description of all aspects of the architecture. Diagrams & narrative text complement each other. Views enhance understanding all aspects of the architecture. Choice of architecture, & decisions about design trade-offs, are well described.	Clear, accurate & mostly complete description of the architecture. Diagrams & narrative text complement each other. Views support description of the architecture. Choice of architecture, & decisions about important design trade-offs, are well described.	Mostly clear, accurate & complete description of the architecture. Diagrams & narrative text support each other. Views support some description of the architecture. Choice of architecture, & decisions about most important design trade-offs, are adequately described.	Fairly clear, & mostly accurate & complete, description of the architecture. Diagrams & narrative text are consistent. Views provide little support describing the architecture. Choice of architecture & decisions about some important design trade-offs, are fairly adequately described.	Some parts of the description are unclear, inaccurate or incomplete. Most diagrams are relevant to the narrative text or a necessary diagram is missing. Justification of choice of architecture is unclear. Decisions about a few important design trade-offs are fairly adequately described.	Some parts of the description are inaccurate or incomplete, or many parts are unclear. Some diagrams are relevant to the narrative text or a few necessary diagrams are missing. Poor justification of choice of architecture. Few design trade-offs are adequately described.	Many parts of the description are unclear, inaccurate or incomplete. Few diagrams are relevant to the narrative text or many necessary diagrams are missing. No, or very poor, justification of choice of architecture. Trade-offs are poorly described.
Architecture Evaluation 20%	Critique & evaluation clearly demonstrate that the delivered architecture, varied a little by the reflection comments, can deliver all functional & non-functional requirements of the full system.	Critique & evaluation clearly demonstrate that the delivered architecture, varied by the reflection comments, can deliver all functional & non-functional requirements of the full system.	Critique & evaluation demonstrate that the delivered architecture, varied by the reflection comments, can deliver all important functional & non-functional requirements of the full system.	Critique & evaluation demonstrate that the delivered architecture, varied by the reflection comments, can deliver all important functional & non-functional requirements of the MVP & part of the full system.	Critique & evaluation demonstrate that the delivered architecture, varied by the reflection comments, can deliver all important functional & non-functional requirements of the MVP but little of the full system.	Critique & evaluation demonstrate that the delivered architecture, varied by the reflection comments, can deliver some important functional & non-functional requirements of the MVP.	Critique & evaluation demonstrate that the delivered architecture, varied by the reflection comments, is unlikely to deliver most functional or non-functional requirements of the MVP. Or, they are too unclear to determine.

References

[1] R. Thomas, "Architectural decision records," February 2023. https://csse6400.uqcloud.net/handouts/adr.pdf.

[2] R. Thomas and B. Webb, "Architectural views," February 2023. https://csse6400.uqcloud.net/handouts/views.pdf.