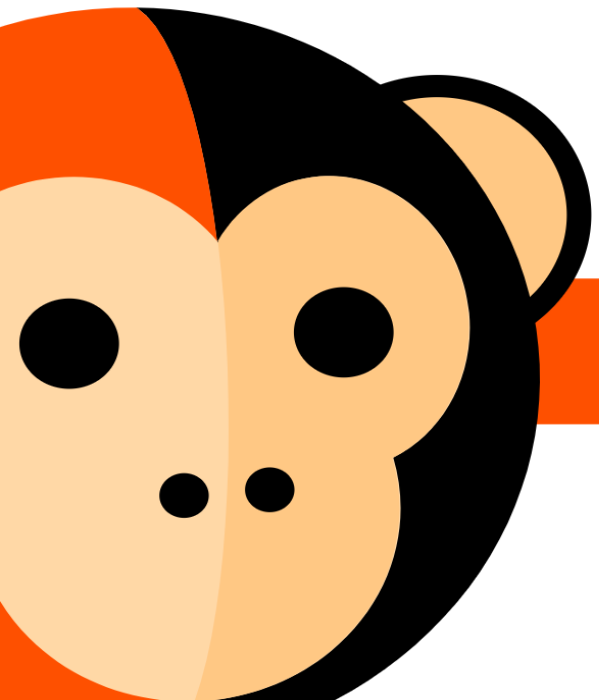
The Practical Architecture Process

How to implement a lightweight, pragmatic way of working for Software Architecture: decision making, design process, traceability, etc.



Moisés Macero García

ThePracticalDeveloper.com

Practical Software Architecture

A Pragmatic way of working for Software Architecture including Decision Making, Design, Traceability and more.

Moisés Macero

This book is for sale at http://leanpub.com/practical-software-architecture

This version was published on 2019-10-11

Contents

1. Introduction

The Practical Architecture Process

Having good ways of working for Software Architecture saves you a lot of time, and helps build the foundations of a successful Software Project. But, how to set up a process that is at the same time efficient, easy to adopt and implement and compatible with Agile methodologies? That's what you'll learn with this book.

First, let's start with the basics: what is a process?

> A process is a set of recurrent or periodic activities that interact to produce a result.
>
> Wikipedia: Process[1]

A typical process flow

How does that definition apply to Software Architecture? That depends on how we define ***result***. In most organizations, the results they expect from Software Architects are:

- Software that is flexible, scalable, modular, fast, etc. The specific list of System Quality Attributes[2] for your project may of course vary.
- A Vision, that shows the long-term technical shape of the project.
- Experience in applying patterns and techniques to solve problems.
- Coaching others, so the knowledge is spread among the team members.
- Creating, and especially helping others create, new ideas that may provide a competitive differentiation to the project.

[1] https://en.wikipedia.org/wiki/Process
[2] https://en.wikipedia.org/wiki/List_of_system_quality_attributes

The better an Architect is at doing those tasks, the better the produced Software Architecture of the project is. Therefore, **the goal as Software Architects is to perform those tasks as good as we can**.

Note that the documents conforming an Architecture Guide are not included in the list of expected results. You will see this remark a few times along the book: documentation is not a goal, it's just an instrument to achieve the goals. The Architecture Documentation is a set of generated *artifacts* from your daily activities. Given that it's one of the most important instruments you have, this book will teach you how to produce this documentation and how to structure it.

So, how to get there? Can we come up with a ***set of activities*** to produce these results and achieve our goals? That's the Software Architecture Process, and this book helps you set up your own one based on a series of steps.

As a difference from a industrial process, you can't manufacture Software Architecture. The purpose here is different. The steps in this book help you **focus on the creative side** and forget about all the *boilerplate* tasks: meeting structure, preparation, agenda, input and output documents, etc.

Why do you need a Software Process?

Let's cover what I consider the three most important reasons.

Reason 1: Your way of working does not work

In this case, your *daily activities* intended to achieve a good Software Architecture don't work. There might be a few reasons for that.

No process at all

A clear evidence of this is when there is scarce knowledge in your organization about how your software is structured, how it behaves and where you want to go with it. Another potential symptom of not counting with a good process is having a *Meeting-Only Software Architect*, that only participates in the software creation process with design meetings, which outcome does not include an action plan and which contents are normally forgotten in a week. A third possibility is having *Reactive Software Architects* that are only summoned when there are problems, and then it's usually too late.

Even though the Software Architects may have a perfect picture of the entire architecture landscape in their minds, for the developers there is a feeling of chaos and the odds are that you deviate from the decisions and conventions very often.

Fake "Agile" Architecture Process

This is a vague term, but here I use it to define the situation of not having any process at all and pretending that it's the best way to go since your company is *Agile*.

In this scenario, you often hear things like "the code is our documentation" and "we don't do diagrams since they become obsolete very soon". Same as the first case, this strategy might work for small teams when combined with a very good communication between its members but, as the team grows, discrepancies and the fact that there are no rules nor conventions may introduce a lot of noise in the daily work.

Heavy (and boring) process

On the other side, there are also situations in which the architects are trapped inside an endless Architecture Process in which everything the team does has to be paired to a formal document, no matter if it's UML, plain text or even a proprietary format. The problem here is twofold:

- The documentation produced and the process itself are seen as a set of forced steps required to continue working on the project, so people may come up with a way to *cheat the system* and do the bare minimum to satisfy the organization. Normally, that ends up with useless documentation for humans.
- The process generates so much output (too verbose) that the developers either don't have time to read the docs or experience *Information Overload*: they have so much information that it's actually harder to understand the foundations and to make decisions about the system (see Information Overload[3]).

As you can see, this is yet another example that extremes are wrong: you can find problems with and without a Software Architecture process. What I aim to achieve with this book is that you find a sweet spot in between: you can configure your own lightweight Process, try it, and adapt it until it works for you.

Reason 2: You keep repeating "boilerplate tasks"

In Software Architecture, a good process can help us remove all the extra effort required for repetitive tasks such as:

- Where do I place this design document? What should I include there? Which tools do I have?
- How many tests does this code require? Do I need to document it? Who do I need to ask to include this extra library?
- Who should I invite for a design meeting? Do we really need a meeting? What should I include on the agenda?

[3] https://en.wikipedia.org/wiki/Information_overload

- I have a problem to solve in code that looks like a boilerplate task, do we have conventions to solve it? Can I reuse a common pattern?

Sometimes we don't notice how much time takes, as a whole group, thinking about this kind of questions. However, when you agree on these basic rules and patterns, you can focus on real, creative and fulfilling work. The reasons are simple:

- Everybody knows where to find the outcome from previous actions (e.g. docs), and they can reuse it.
- You avoid endless, recurrent discussions about trivial topics (see next section about making decisions).
- Since everything is organized the same way, you don't need to look into multiple places or ask multiple people to find what you're looking for.

Reason 3: You don't have a good decision-making mechanism

I've seen many times teams that struggle to make decisions related to their Software Architecture. Other teams make decisions very fast, but nobody follows them later. There is no single reason behind those problems, it's like a *cocktail* with several possible ingredients:

- **The Dictator**. Only one person makes the decisions without consulting others. In this case, it doesn't matter if the outcome is documented or not. It's unlikely that the decisions are good since they're never taking into account multiple perspectives. Besides, the team won't feel represented so they won't embrace those decisions. The result is a team that is either feeling frustrated for following rules that they don't understand or skipping the rules because they feel the decisions don't make sense.
- **The Decision *Queue with multiple Workers***. The first person who faces a common problem defines the common solution. This is a variation of the first one with same results. If that person is not taking into account other requirements by involving others, the decisions won't be good.
- **The Déjà vu**. With this one, we enter into the world of making decisions involving a group of people. The déjà vu happens when you come back to problems that you've seen before, and you're making decisions about a specific topic that was already discussed in the past. There are two variations of the déjà vu:
  - You discussed the problem in the past but you never made a decision on how to solve it. Normally, this is manifested as a series of meetings that go nowhere, in which the attendees tend to focus too much on how bad the problem is, but not on thinking of possible solutions and defining actions.
  - You discussed the problem in the past, and you made a decision, but there you are again… It might be caused by a badly documented decision or bad communication (a decision was made but nobody knows it). The reason can also be that some people don't like the decision and would like more time to argue about that.

- **The Tabs vs. Spaces fights**. Trivial discussions are devouring your working time. You enter a meeting with the goal of making a decision that is important for the project but somehow you end up consuming most of the time in topics that are minor and should be decided quickly (they can be easily changed later and/or there is no clear preference for one option). Some of these trivial topics, micro-problems attacking your meetings, may also be recurrent (let's call them *micro déjà vu's*).

Having a good process in place will help you deal with these situations and come up with productive outcomes. Surely it won't remove these issues from the working environment since they just happen to be part of software development, but a pragmatic process can make a big difference.

The Process Stigma: sounds boring

The word *process* may trigger a negative feeling. It's normal: many of us have suffered from extremely boring tasks that were required during the normal software development process.

In the past, I've been required to fill in a document of about ten pages each time we wanted to include a new library as a dependency in our Java project. Even a popular one, such as *Guava* or *Apache Commons Lang*. It could take almost one day to fill in the document, plus a few weeks for the document administration to finish. Say four weeks are required before you can use that useful method that will save you write 200 extra lines of code. If you're under pressure, you'll forget about the library and put those extra lines in your code, paying a higher price in maintenance and *code crappiness*. In this case, the process is not only boring but annoying and counter-productive.

You don't need an ISO norm, going too formal or hiring extra people to implement a process. All you need is a Git repository or a wiki-style platform where you can write the steps to follow to achieve something or the decisions you've been making over time. And, as you'll see in the next sections, you can create a lightweight process, tailored to your needs and easy to maintain and follow.

A good process is like a good recipe. Maybe you copied that recipe from a website and it's too detailed and it takes too long to read. Maybe you don't like spices so you remove them from the process. Then, you modify the recipe and make a list of steps that work for you, that makes you achieve that dish that you love: a great outcome. After some time, you don't even need to follow the recipe because you know it by heart. But you got there by taking notes and fixing the process, and that's much more efficient than improvising the steps each time you prepare the dish and trying to remember them the next time.

What to expect from this book?

In the next chapter, you learn how the set of artifacts produced by the process, the Architecture Guide, helps you keep a nice Software Architecture by making the knowledge accessible to others

and avoiding you to waste time in repetitive discussions. Even though it was already mentioned that the Guide is not your goal, you should know the structure to better understand the inputs/outputs detailed in the next chapters.

Then, in chapter 3, you find the different process' pieces explained in a simple way. There is an important part describing how you can write down effective Decision notes.

Chapters 4 to 6 are the core of the book and explain the process in detail, describing how you can approach meetings, documentation and design sessions so you can get the best result of them.

The last Chapter is to wrap up with conclusions and you'll find an extra Appendix that explains the different Architecture Guide sections in detail.

2. Process Outcome: The Architecture Guide

Introduction

The Architecture Guide is an alive, evolving set of documents. It's a result of the Process and holds the designs, conventions, and decisions you make.

Both the Guide and the Process are important so you shouldn't focus only on one of them. If you get a perfect Architecture Guide without a good process supporting it, that will be like giving a book to the development team and hoping that they'll follow it without questions; that's a top-bottom approach unlikely to succeed. On the other hand, if you have a very efficient process but the results are not properly organized and documented, you'll experience the *déjà vu* effect.

The owner and ultimate responsible person for keeping it up-to-date should be the Software Architect(s), but the contributors might be many. Think of the Architect as an editor reviewing and organizing pages created by developers, giving consistency and providing a high level of structure. Usually, technical people don't like to write documentation so it's hard to get others to contribute; anyhow, you should encourage it.

Templates

The easiest way to get a well-structured Architecture Guide from scratch is using one of the templates available online.

The main advantage of using a template is that you don't need to come up with your own structure. Also, a template helps you remember all contents that you should cover at some point.

On the other hand, having a blank template may put a lot of pressure on architects to start writing docs and have it completed as soon as possible. In any case, if you base your Architecture Guide on a template from any of the sources listed below, try to avoid having blank sections. Just omit them until you have something to tell under that section. A guide full of *to-be-completed* tags makes difficult to find the relevant contents.

Remember that the process described in this book will help you build your guide step by step so don't hide in a corner and start typing on your own until you have it completed; that won't work as you would expect.

No matter which template you use, always feel free to modify it as you wish. Don't try to write some meaningless content in some sections just because you want to maintain a given structure.

Software Architecture for Developers

This Architecture Guide's template is explained in detail in the book Software Architecture for Developers (Volume II)[4] by Simon Brown. In that practical book, you'll see a detailed description of all the sections that you should include and advice on how to keep it focused on what's important. It also covers the C4 model, a very useful way to visualize Software Architecture. These are the headings for the Guide proposed in the book:

1. Context
2. Functional Overview
3. Quality Attributes
4. Constraints
5. Principles
6. Software Architecture
7. Code
8. Data
9. Infrastructure Architecture
10. Deployment
11. Operation and Support
12. Development Environment
13. Decision Log

Along with every section's description, you'll find in the book its intent, structure, motivation and audience.

Arc42 Template

This template, created by Gernot Starke and Peter Hruschka, is available online at arc42.org[5]. It has a Creative Commons license[6] and is fully available online for your reference.

The sections proposed in that template are the following:

1. Introduction and Goals
2. Architecture Constraints
3. System Scope and Context
4. Solution Strategy
5. Building Block View
6. Runtime View
7. Deployment View

[4] https://leanpub.com/visualising-software-architecture
[5] https://arc42.org
[6] https://arc42.org/license

8. Cross Cutting Concepts
9. Design Decisions
10. Quality Requirements
11. Risks and Technical Debts
12. Glossary

Each of these sections count with a page with details explaining what you should include there, and the reasoning behind it.

The Practical Developer's Template

As you could expect, I also have my own template that I've been evolving over the last years.

1. Guide
    1. Functional
    2. Conceptual Model
    3. Architecture Definition (Vision, Constraints, Risks, Principles)
    4. Architecture Overview (overall solution and diagrams)
    5. Modules (and how they map to Teams)
    6. Common Patterns
    7. Infrastructure
    8. Integration and Deployment
    9. Operations
2. Roadmap
    1. Architecture Goal
    2. 2019-Q3 Planning
    3. 2019-Q4 Planning
    4. ...
3. Designs
4. Decisions
5. Journal
6. Modules

In this case, I prefer keeping contents that will change very often (sections 2-6) apart from the main Guide sections, which are normally edited less frequently.

This book helps you build a Guide based on this template. Over the next chapters, you'll see how following simple steps will lead you to have effective documentation. If you prefer to use any other template, most of the sections can be mapped anyway.

You can find more details about each section on the Appendix I of this book. A reference Guide structure with descriptions is also available on Github at The Practical Architecture Guide Repository[7].

The Guide is not your Goal

All the guide details are at the end of the book so you don't lose the focus from the main topic: the Process. The Architecture Guide is a very useful set of Artifacts but it's not your Goal as a Software Architect.

Your objective is building the Software Architecture your project needs. To achieve that, you have to follow some steps: think, design, talk to people and go to meetings, amongst others. As a result, you produce diagrams, conclusions, decisions, etc. When you write them down and you put them together, you have the Architecture Guide.

You shouldn't aim to only improving the way you write documentation or its structure. You should focus on having a good process, on following the proper steps that not only will lead you to produce good documentation but also to your success as a Software Architect.

Documentation Platform

To write and maintain the Guide, you can use any kind of document sharing tool already available in your company and set up a dedicated space with these sections as placeholders. Make sure that all project members can see and edit. It should provide history of changes so you know what has been modified in a page, when and by whom.

I recommend you, however, to **use a Git repository for the Guide** and all other related documents. Then, you can use a lightweight markup language such as Markdown[8] to evolve docs and register decisions and new designs.

My main reasons for recommending a Git approach are not the editing features of a markup language or the ease to maintain of a Git repository. To me, the most important advantage you'll get is that **you'll make easier and more attractive for your contributors to keep the docs up-to-date**. These docs are written by technical people and mainly intended for technical people. Developers would rather download the Guide repository in their preferred IDE and start updating the code (Markdown for example) instead of navigating through the organization's intranet.

For the rest of your readers, you can make all these docs look nice too, and you can publish them in a format that is accessible by everybody. For example, an HTML site.

If you go for the Git approach, remember that you can use The Practical Architecture Guide Repository[9] as a starting point.

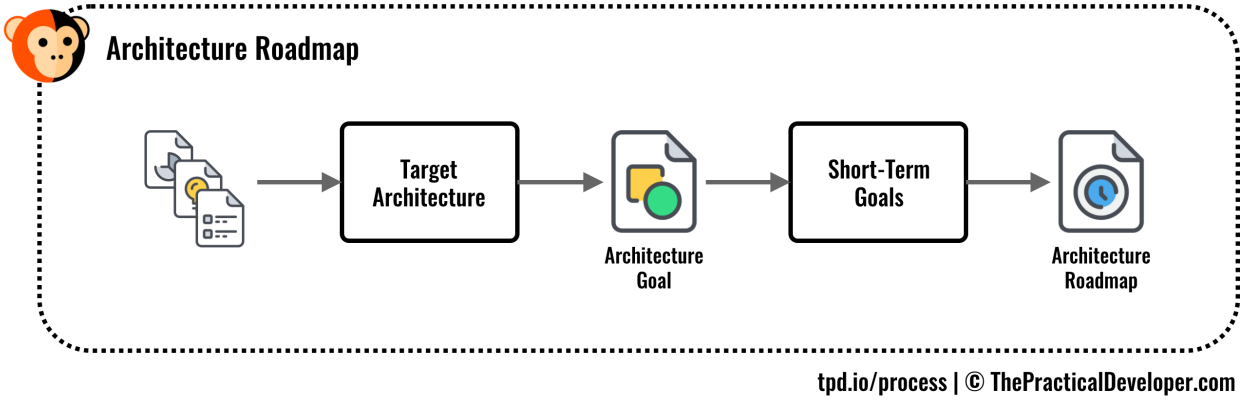
[7] https://github.com/thepracticaldeveloper/practical-architecture
[8] https://en.wikipedia.org/wiki/Markdown
[9] https://github.com/thepracticaldeveloper/practical-architecture

Legacy Documentation

Link to your Guide any existing documentation that you already have and that you consider useful. However, do this only if you can keep those docs lightweight and to the point. Try to avoid building a Frankenstein Guide copy-pasting from multiple docs with different styles.

As an alternative, you can also create an extra section in your Guide, *Legacy Docs*, and index there your old documents. Then, you can incrementally build your new Guide using the Process described in this book.

5. Block II: Architecture Roadmap



Architecture Roadmap

Description

After you define the broad topics it's time to get more specific. Some of the most practical sessions you can have in this process are the Target Architecture Definition and the Roadmap Session. During those, you'll focus on the next steps to build the Architecture you're aiming for.

Note that **this part involves a lot of thinking and preparation before going to real meetings**. First, you need to define your target architecture taking into account all the inputs you have available and your experience and experience from others. You need to consider different alternatives and think of their benefits and their disadvantages for your case. Then, you need to define what are the next steps that you should take in order to achieve the target architecture. For both tasks, it's important that you get feedback and input from others so you end up with a good consensus. However, it's also crucial that you prepare all the context in advance so you don't start discussions with a blank page. If you don't set up a frame and context to enable discussions, the situation can be hectic.

Meeting: Target Architecture

Inputs

As mentioned above, this is one of the steps that require more preparation in advance. It consists of one of your main duties as Software Architect: defining the target architecture of the project

based on the requirements, the time available, resources, etc. Use the structure of the Architecture Guide to fill in some contents, **but don't restrict yet to a single option**. You have to explain, for instance, what are the benefits of going for a modular monolith instead of microservices, and also the disadvantages (remember, this is just an example). Try not to copy general conclusions from books or articles but adapt them instead to your specific environment.

Remember that you should cover not only the target software composition of your project but also its infrastructure, the Continuous Integration setup and how Deployment will look like. You don't need to be very specific at this point but it's essential that you are realistic and pragmatic when listing the alternatives. This is not an easy task nor something that a single person can do efficiently so try to do this collaboratively with other people.

Take also into account that you won't be accurate at the beginning. That's not a problem, your Target Architecture may evolve over time after you try things and become aware of unexpected problems or constraints, or you figure out simpler ways of building your Software.

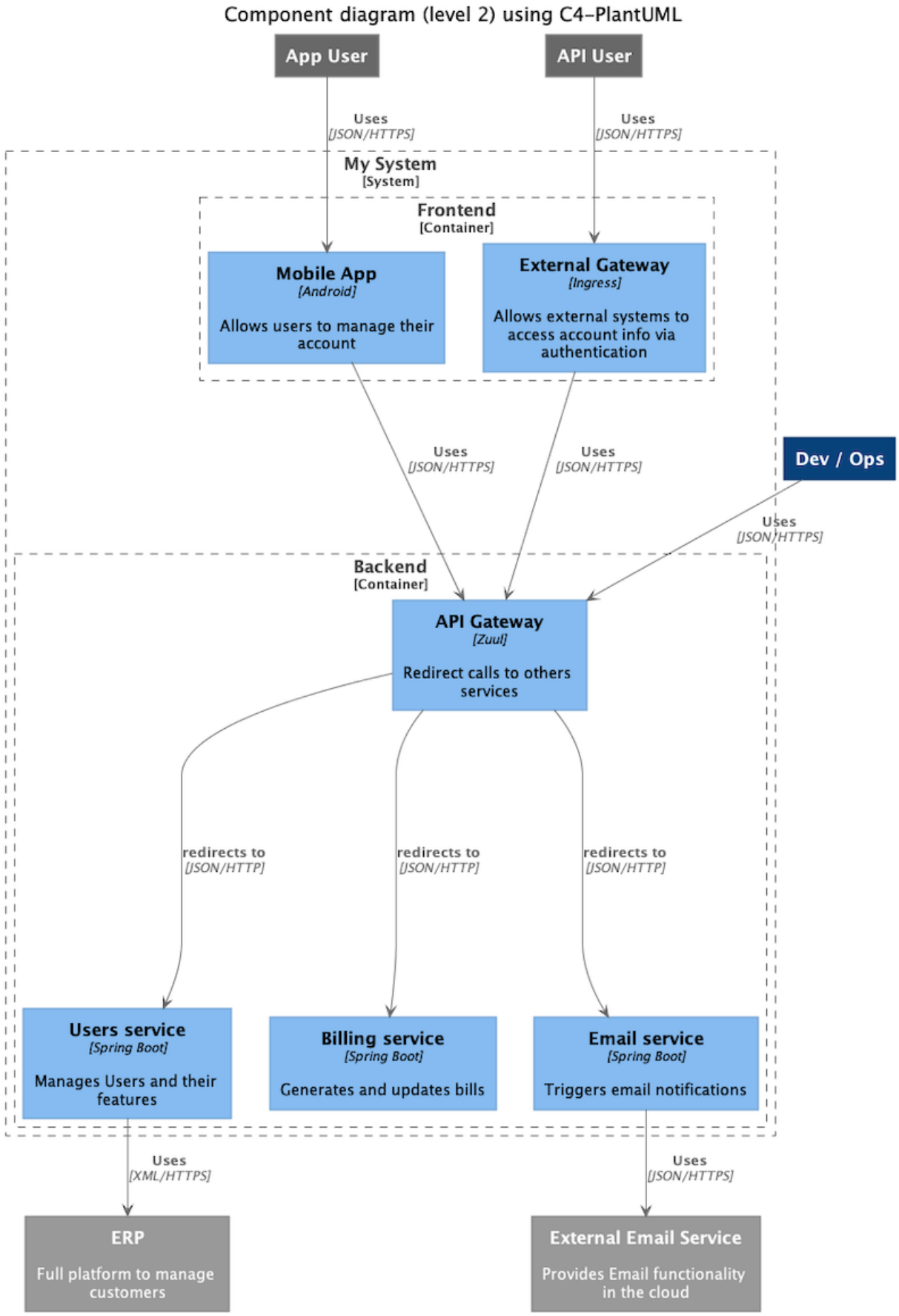
Use **diagrams** to represent the different options and how you plan to organize your components, infrastructure, etc. They are easy to understand and will serve as a base for possible discussions. Again, I recommend you to use the C4 model[10] since it's way simpler and clearer than other documentation methods. If you use this model, prepare draft alternatives for Level 1 (System Context) and 2 (Container) diagrams.

Diagrams as Code with PlantUML

I strongly recommend everybody to use **code** to build diagrams. It's very easy to maintain them and you can modify your diagrams in a few seconds when something changes. And the extra advantage that, being text, you can track changes on those diagrams in the same way you do with code and docs.

My preferred tool is PlantUML[11]. There are also some extensions written by the community that allows you to create some better-looking diagrams, like C4-PlantUML[12]. The diagram below has been created with this extension, and you can find its source code in the book resources repository[13].

[10] https://c4model.com/
[11] http://plantuml.com/
[12] https://github.com/RicardoNiepel/C4-PlantUML
[13] https://github.com/thepracticaldeveloper/practical-architecture/blob/master/1_guide/diagrams/04_architecture_c2_c4puml.puml



Example of Component Diagram using C4-PlantUML

Audience

For this meeting, you need mainly a Technical Audience with expert representatives. However, having at least one person from the pure Business side is very recommended. That person might help you keep the focus on how the different alternatives are directed to build the target requirements.

Format and Duration

Keep the duration of the meeting to 90 minutes maximum. Aim to finish it in one hour. For this meeting, it's extremely important that the attendees receive all the prepared diagrams and descriptions in advance, so they go through them **before the meeting**. Send reminders one or two days before the meeting to make sure they read the docs and prepare their questions.

During the meeting, use the first 15 minutes to go over the different options, their pros, and cons. Express your opinion about what you believe it's the best option and why. Then, because it's almost impossible that your diagrams/documentation are 100% clear, use some time just to answer questions about the different options.

After the questions, leave some time so others can express their concerns and propose other ways to design parts of the system. However, it should be clear before the meeting that, if someone has a completely different approach for the Target Architecture, they should prepare the same diagrams/documentation as you did **before** that session.

In some cases, the discussion may become difficult. For instance, let's say you're proposing to start with a modular monolith and split later when needed. Some people may argue that you might be trapped with a monolith forever, and defend a position of building microservices from the beginning. That's a normal discussion, but try to always keep the conversation around a proper rationale: advantages, disadvantages, constraints and overall plan for your specific situation.

In that same example, the microservices approach should have been analyzed before the meeting, taking into account its impact on continuous integration, deployment, ways of working within teams, etc. Part of your work as Architect is to make people aware of other side aspects of Software Architecture that affect the decisions you need to make: how do you plan to get there? Are there enough developers to work on that? Do we have the proper tools? Does it make sense for our specific use case and requirements? Is it the best option for our long-term vision goals? There will be occasions in which you just need to make a decision and move forward, not to get stuck in repetitive discussions.

Outputs

As usual, you need to write a new entry in the Architecture Journal. Then, you should revisit your Software Architecture page (e.g. diagrams and descriptions) and modify them accordingly, based on the feedback you got and the decisions you made during the meeting.

You may need to repeat this session two or three times until you get to a valid, sufficiently-defined Target Architecture to start with. Don't try to extend the duration of the session to get results sooner since this is the kind of meeting that doesn't go better when it lasts longer.

Take the artifact resulting of this session, the Architecture Goal, and include it as the first heading in the section *Roadmap* of your Guide. Note that, as you move forward and evolve your architecture towards the target, you should also update the diagrams describing your current architecture. Those are better placed under the section *Architecture Overview*. Don't put your target diagrams there; they don't reflect your current status and may confuse people instead of help them.

Meeting: Short-Term Goals

Inputs

First, you need to have the Architecture Goal artifact defined before this session. Having that as your first input, you need to combine it with the Functional Roadmap: what are the features that you're planning to build first? Ideally, that roadmap is a bit flexible and can be modified depending on the Architecture requirements but, even if not, it's a valuable input.

If you're not starting from scratch, you should use as input the list of topics that could significantly change your Architecture for the better. If you don't have anything yet, create a list of tasks that you need to accomplish. Think of this kind of tasks: *automation of build steps to reduce manual interaction, design module boundaries, create (or adopt) code conventions, extract part of common logic to a library, etc.*

Audience

Technical Audience. If the task is not purely about Architecture (for instance designing module boundaries), then invite someone with Business knowledge.

Format and Duration

These sessions should take 60 minutes maximum. Start with the goal of the meeting: to prioritize and pick the next Architecture work items. Present the list of tasks that you already identified and ask others to contribute to the list.

Try to use a whiteboard and sticky notes to allow for better organization. First, let everybody list what they think that are the most important tasks to do. Then, use dot-voting[14] to identify what are the most valuable. After everybody has voted, organize the stickers in three columns that represent how easy those tasks are in terms of complexity and time: *easy, medium, hard*. Decide where to place each task as a group.

[14] https://en.wikipedia.org/wiki/Dot-voting

Once you have all the tasks in the three columns and with their respective dots, it should be easy to identify the quick wins: all of them with a high score placed in the "easy" column. Anyway, you can't forget all the other ones, especially those with many dots. Try to pick one of the high-scored hard ones at a time. Don't forget to take a picture of the whiteboard so you can upload it later to the session notes.

Agree on the first 3 to 5 topics to work on and then assign (as a team) action points: who can work on what? If they're not ready to be picked but they need some further analysis, who can take care of that? As usual, write these action points down.

The frequency of these sessions depends on your needs, but it may be good to start with one per month and then adapt to once per quarter.

Outputs

After the session, write down the meeting notes including the action points and the whiteboard picture. Plan the follow-up session so you don't forget about it. Add a new entry to the Architecture Journal pointing to those meeting notes. Share it, and ask for possible feedback.

Now you need to coordinate with Project Managers (or Product Owners) how those items are going to be properly refined and taken inside the Sprints (or any other unit of planning that the teams have, in case you're not doing Agile). Some of these items are purely technical (for instance, let's say you want to extract all the logs to a common aggregator), yet it's vital that they are part of the planning together with other feature-related stories. If you don't manage to find a good balance between them, your project will most likely end up with no proper architecture at all, just a bunch of software blocks that at some point will become unmaintainable and very expensive to evolve.

Once you have agreements on planning, fill in the Roadmap section with a simple map of tasks/calendar/teams. Use your preferred format, but a table like shown below can be just good enough.

Task	Sprint 1	Sprint 2	Sprint N
Extract Email module	Team A	Team A	
Reuse Kafka component		Team B	

Instead of editing the same Roadmap along time, keep a separate version per quarter (or month) as proposed in the Guide structure. That will give you a quick historical reference and you can also add some notes later if you find impediments or obstacles.