Community Experience Distilled

Software Architecture with Python

Architect and design highly scalable, robust, clean, and highly performant applications in Python

Anand Balachandran Pillai

[PACKT] open source
PUBLISHING
community experience distilled

Table of Contents

1
Principles of Software Architecture

This is a book on Python. At the same time it is a book about Software Architecture – and its various attributes which are involved in software development life cycle.

In order for you to understand and combine both aspects together – which is essential to get maximum value from this book, it is important to grasp the fundamentals of Software Architecture, the themes and concepts related to it, and the various quality attributes of Software Architecture.

A number of software engineers who are taking on senior roles in their organizations often get very different interpretations of the definitions software design and architecture and the roles they play in building testable, maintainable, scalable, secure, and functional software.

Though there is a lot of literature in the field that is available both in conventional book forms and on the Internet – very often the practitioners among us get a confusing picture of these very important concepts. This is often due to pressures involved in "learning the technology" rather than learning the fundamental design & architectural principles underlying the use of technology in building systems. This is a common practice in software development organizations where the pressures of delivering working code often overpowers and eclipses everything else.

A book such as this one, strives to transcend the middle path in bridging the rather esoteric aspects of software development related to its architectural quality attributes to the mundane details of building software using programming languages, libraries and frameworks – in this case using Python and its developer ecosystem.

The role of this introductory chapter is to demystify these concepts and explain them in very clear terms to the reader so as to prepare him on the path towards understanding the

rest of this book – so that towards the end of this book, the concepts and their practical details – would represent a coherent body of knowledge to the reader.

We will then get started on this path without any further ado, roughly fitting into the following sections.

- Defining Software Architecture
- Software Architecture vs Design
- Aspects of Software Architecture
- Characteristics of Software Architecture
- Why is Software Architecture Important
- System vs Enterprise Architecture
- Architectural Quality Attributes
    - Modifiability
    - Testability
    - Scalability/Performance
    - Security
    - Deployability
- Summary

Defining Software Architecture

There are various definitions of Software Architecture in the literature. Wikipedia defines Software Architecture as,

"Software architecture refers to the high level structures of a **software** system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the **software** system."

A more formal definition, from the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE 1471) is,

> *"Architecture is the fundamental* **organization** *of a* **system** *embodied in its* **components**, *their* **relationships** *to each other, and to the* **environment**, *and the principles guiding its design and evolution."*

It is possible to get umpteen such definitions of Software Architecture if one spends some time searching on the Web. The wordings might differ but all the definitions refer to some core, fundamental aspects underlying Software Architecture.

Software Architecture vs Design

In the author's experience, this question of Software Architecture of a system vs its Design seems to pop-up quite often, in both on-line as well as off-line forums. Hence let us take a moment to understand this aspect.

Though both terms sometimes are often used interchangeably, the rough distinction of architecture vs design can be summarized as,

- Architecture is involved with the higher level of description structures and interactions in a system. It is concerned with those questions that entail decision making about the *skeleton* of the system involving not only its functional but also its organizational, technical, business and quality attributes.

  We will see below the various aspects of Software Architecture.

- Design is all about the organization of parts or components of the system – and the sub-systems involved in making the system. The problems here are typically closer to the code or modules in question such as,
  - What modules to split code into? How to organize them?
  - Which classes (or modules) to assign the different functionalities to?
  - Which design pattern I should use for class "C"?
  - How do my class objects interact at runtime? What are the messages passed and how to organize the interaction?

Software Architecture is about the design of the entire system, whereas Software Design is mostly about the design and details, typically at the implementation level of the various sub-systems and components that make up those sub-systems.

In other words the word "design" comes up in both contexts – however with the distinction that the former is at a much higher abstraction and at a larger scope, than the latter.

There is a rich body of knowledge available for both Software Architecture and Design – namely Architectural Patterns and Design Patterns respectively. We will discuss both these topics in later chapters in this book.

Aspects of Software Architecture

In both the formal IEEE definition and the rather informal Wikipedia definition earlier, we find some common themes or aspects that recur. It is important to understand them in order to take our

discussion on Software Architecture further.

- **System –** A system is a collection of components organized in specific ways to achieve a specific functionality. A software system is a collection of such software components. A system can often be sub-grouped into sub-systems.
- **Structure –** Structure is a set of elements that are grouped or organized together according to a guiding rule or principle. The elements can be software or hardware systems. A software architecture can exhibit various levels of structures depending on the observer's context.
- **Environment -** The context or circumstances in which a software system is built, which has a direct influence on its architecture. Such contexts can be technical, business, professional, operational etc.
- **Stakeholder** – Anyone, person or groups of persons, who has an interest or concern in the system and its success. Examples of stakeholders are architect, development team, customer, project manager, marketing team etc.

Now that you have understood some of the core aspects of Software Architecture, let us briefly list what are some of the characteristics of Software Architecture.

Characteristics of Software Architecture

All software architectures exhibits a number of common set of characteristics. Let us look at some of the most important ones here.

An architecture defines a structure

An architecture of a system is best represented as structural details of the system. It is a common practice for practitioners to draw the system architecture as a structural component or class diagram in order to represent the various relationship between the sub-systems.

For example, the following architecture diagram describes the back-end of an application that reads from a tiered database system, which is loaded using an ETL process.

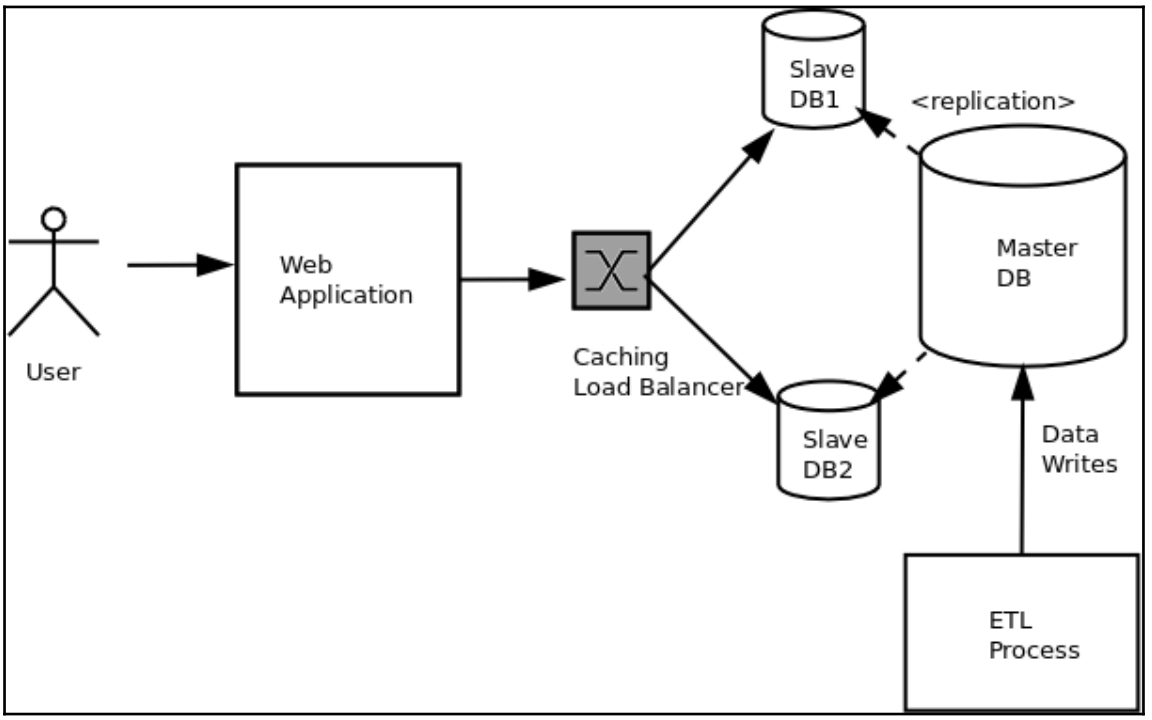

Figure 1. Example Architecture Diagram showing system structure

Structures provide insight into Architectures and provides a unique perspective to analyze the architecture with respect to its quality attributes.

Some examples are:

- The runtime structures, in terms of objects created at run-time and how they interact often determine the deployment architecture. The deployment architecture is strongly connected to the quality attributes of scalability, performance, security and interoperability.
- The module structures, in terms of how the code is broken down and organized into modules and packages for task break-down, often has a direct bearing on the maintainability and modifiability (extensibility) of a system. For example,
- Code which is organized with a view of extensibility would often keep the parent classes in separate well identifiable packages with proper documentation and configuration, which are then easily extensible by external modules, without needing to resolve too many dependencies.

- * Code which is dependent on external or third-party developers (libraries, frameworks etc), would often provide setup or deployment steps which manually or automatically pull in these dependencies from external sources. Such code would also provide documentation (README, INSTALL etc) which clearly document these steps.

An architecture picks a core set of elements

A well-defined architecture clearly captures only the core set of structural elements required to build the core functionality of the system and which have a lasting effect on the system. It does not set out to document everything about every component of the system.

An architecture captures early design decisions

This is a corollary to the characteristic described previously. The decisions that helps an architect to focus on some core elements of the system (and their interactions) are a result of the early design decisions about a system. Thus these decisions play a major role in further development of the system due to their initial weight.

An architecture manages stakeholder requirements

A system is designed and build ultimately at the behest of its stakeholders. However, it is not possible to address each stakeholder requirement to its fullest due to often contradictory nature of such requirements. Examples are:

- The marketing team is concerned with having a full featured software application, whereas the developer team is concerned with "feature creep" and the performance issues when adding a lot of features.
- The system architect is concerned with using the latest technology to scale out his deployments to the cloud while the project manager is concerned about the impact such technology deployments will have on his budget.
- The end user is concerned about correct functionality, performance, security, usability and reliability while the development organization (architect, development team + managers) is concerned with delivering all these qualities, while keeping the project on schedule and within budget.

A good architecture tries its best to balance out these requirements by making trade-offs,

delivering a system with good quality attributes while keeping the people and resource costs under limits.

An architecture also provides a common language, a kind of lingua franca – among the stakeholders which allow them to communicate efficiently, via expressing these constraints and helping the architect zero-in towards an architecture that best captures these requirements and their trade-offs.

An architecture influences organizational structure

Quite often the system structures an architecture describes have a direct mapping to the teams that build those systems.

For example, an architecture may have a data access layer which describe a set of services that read and write large sets of data – it is natural that such a systems gets functionally assigned to the Database team which already has the required skill sets.

Since the architecture of a system is its best description of the top-down structures, it is also often used as the basis for the task-breakdown structures. Thus software architecture has often a direct bearing on the organizational structures that build it.

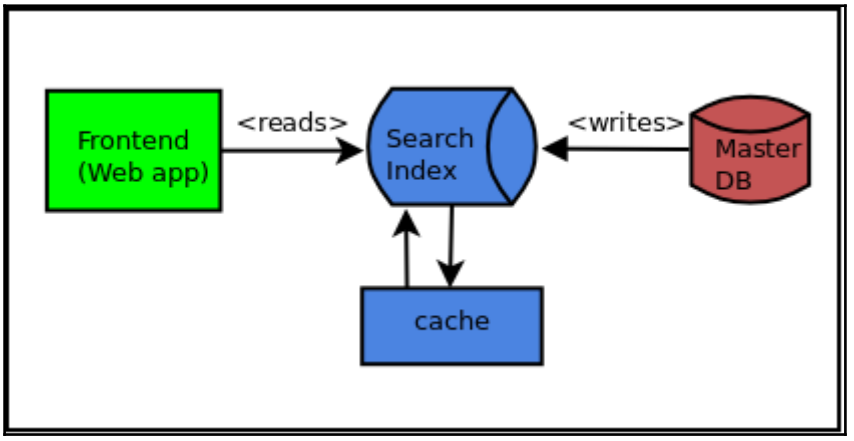

Figure 2. System architecture for a search web application

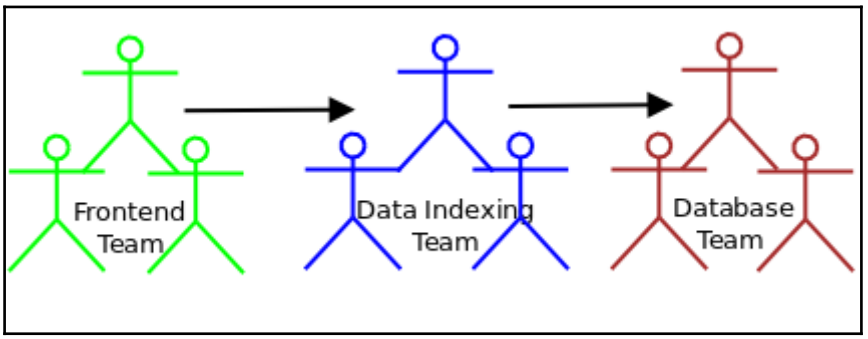

Figure 3. Possible team structure for the search web application

An architecture is influenced by its environment

An environment imposes outside constraints or limits within which an architecture must function. These are often called in literature as “architecture in context” [Ref: Bass, Kazman]. Some examples are:

- Quality attribute requirements - In modern day web applications, it is very common to specify the scalability and availability requirements of the application as an early technical constraint and capture it in the architecture. This is an example of a technical
context from business perspective.
- Standards conformance - In some organizations, especially those in the banking, insurance and health-care domains, where there is often a large set of governing standards for software, these get added to the early constraints of the architecture. This is an example of an external technical context.
- Organizational constraints - It is common to see organizations which either have an experience with a certain architectural style or a set of teams operating with certain programming environments that impose such a style (J2EE is a good example), preferring to adopt similar architectures for future projects as way to reduce costs and ensure productivity due to current investments in such architectures and related skills. This is an example of an internal business context.
- Professional context - An architect's set of choices for a system's architecture, aside from these outside contexts, is mostly shaped from his set of unique experiences. It is common for an architect to continue using a set of architectural choices that he has had most success in his past, for new projects.

Architecture choices also arise from one's own education and professional training and also from the influence of one's professional peers.

These are all examples of architectural environments in a professional context.

An architecture documents the system

Every system has an architecture, whether it is officially documented or not. However, properly documented architectures can function as an effective documentation for the system. Since an architecture captures the system's initial requirements, constraints and stakeholder trade-offs, it is a good practice to document it properly. The documentation can be used as a basis for training later on. It also helps in continued stakeholder communication and for subsequent iterations on the architecture based on changing requirements.

An architecture often confirms to a pattern

Most architectures confirm to certain standard set of styles which have had a lot of success in practice. These are referred to as architectural patterns
. Examples of such patterns are Client-Server, Pipes and filters, Data based architectures etc. When an architect choses an existing pattern, he gets to refer to and reuse a lot of existing use-cases and examples related to such patterns. In modern day architectures, the job of the architect comes down to mixing and matching existing set of such readily available patterns to solve the problem at hand.

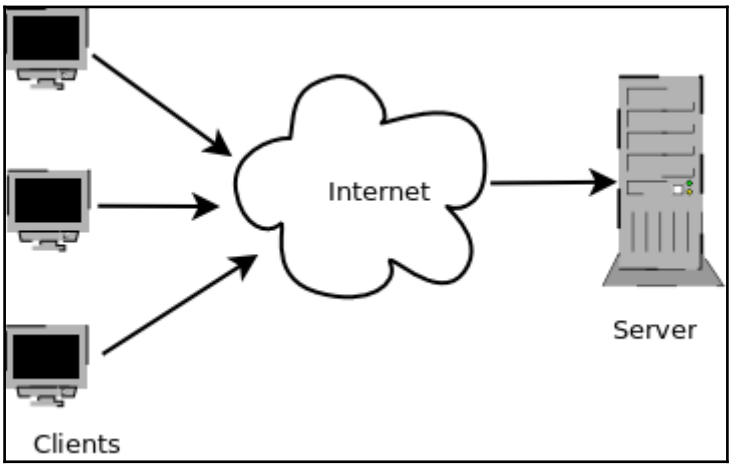

Figure 4: Example of Client-Server Architecture

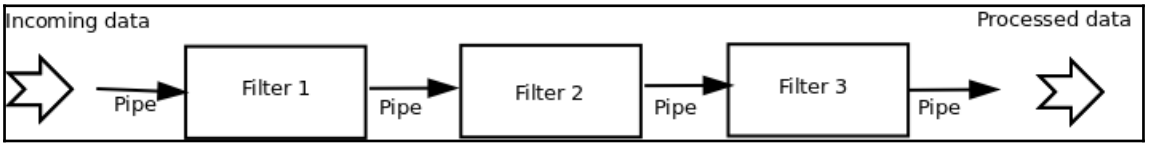

Figure 5: Example of Pipe and Filters Architecture

We will see examples of architectural patterns later in this book.

Why is Software Architecture Important

So far, we have discussed the fundamental principles of software architecture and also seen some of its characteristics. These sections of course assumed that Software Architecture is important and is a critical step of the software development process.

It is time to play devil's advocate and look back at Software Architecture and ask some existential questions to it!

- Why Software Architecture? Why is it important?
- Why not build a System without a formal Software Architecture?

Let us take a look at what are those critical insights software architecture provides which

would otherwise be missing from an informal software development process. We are only focusing on the technical or developmental aspects of the system in the below list.

Aspect	**Insight/Impact**	**Examples**
Architecture selects quality attributes to be optimized for a system	Aspects like Scalability, Availability, Modifiability, Security etc of a system depends on early decisions and trade-offs while selecting an Architecture. You often trade one attribute in favor of another.	A system that is optimized for scalability must be developed using a de-centralized architecture where elements are not tightly coupled. E.g: Micro-services, Brokers.
Architecture facilitates early prototyping	Defining an architecture allows the development organization to try and build early prototypes which gives valuable insights into how the system would behave without having to build the complete system top down.	Many organizations build out quick prototypes of services – typically by building only the external APIs of these services and mocking the rest of the behavior. This allows for early integration tests and figure out interaction issues in the architecture early on.
Architecture allows a system to be built component-wise	Having a well-defined architecture allows to reuse and assemble existing readily available components to achieve the functionality, without having to implement everything from scratch.	Libraries or frameworks which provide ready-to-use building blocks for services. E.g: Web application frameworks like Django/RoR, Task distribution frameworks like Celery.
Architecture helps to manage changes to the system	An architecture allows the architect to scope out changes to the system in terms of components that are affected and components which are not. This helps to keep system changes to a minimum when implementing new features, performance fixes etc.	A performance fix for database reads to a system would need changes only to the DB and DAL (data access layer) if the architecture is implemented correctly. It need not touch the application code at all. For example, this is how most modern web frameworks are built.

There are a number of other aspects which are related to the business context of a system, which architecture provides valuable insights to. However, since this is a book mostly on the technical aspects of software architecture, we have limited our discussion to the above.

Now, let us take on the second question "*Why not build a system without a formal Software Architecture ?*"

If you've been following the arguments so far thoroughly, it is not very difficult to see the answer for it. It can though be summarized in a few statements.

- Every system *has* an Architecture, whether it is documented or not.
- Documenting an Architecture makes it formal, allows it to be shared among stakeholders and makes change management and iterative development possible using it.
- All the other benefits and characteristics of Software Architecture are ready to be taken advantage of, when you have a formal architecture defined and documented.
- You may be still able to work and build a *functional* system without a formal architecture, but it would not produce a system which is extensible and modifiable and would most likely produce a system with a set of quality attributes quite far away from the original requirements.

System vs Enterprise Architecture

You may have heard the term Architect used in a few contexts. The following job *roles* or *titles* are pretty common in the software industry for architects.

- Technical Architect
- Security Architect
- Information Architect
- Infrastructure Architect

You also may have heard the term *System Architect* and perhaps the term *Enterprise Architect* and maybe also *Solution Architect* . The interesting question is, "*What do these people do ?*"

The short answer is:

1. An Enterprise Architect looks at the overall business and organizational strategies for an organization and applies architecture principles and practices to guide the organization through the business, information, process, and technology changes necessary to execute their strategies.

   The Enterprise Architect has a higher strategy focus and a lower technology focus usually.

2. The other Architect roles take care of their own sub-systems and processes. For example:
    1. A Technical Architect is concerned with the core technology (Hardware/Software/Network) used in an organization
    2. A Security Architect creates or tunes the security strategy used in applications to fit the organizations information security goals
    3. Information Architect comes up with architectural solutions to make information available to/from applications in a way that facilitates the organization's business goals.

    These specific architectural roles are all concerned with their own systems and sub-systems. So each of these role is a *System Architect* role.

    These architects – in a large organization there may be several – help the Enterprise Architect to understand the smaller picture of each of the business domain they are responsible for, which helps the Enterprise Architect to get information that will aid him in formulating business and organizational strategies.

    A System Architect usually has a higher Technology focus and a lower Strategy focus. Also his work is more at the scope of projects within the organization (Project Scope) than at the organizational level itself (Organizational Scope).

3. Sometimes it is a practice in service oriented software organizations to have a *Solution Architect* role who combines the different systems to create a solution for a specific client. In such cases the different Architect roles is often combined into one, depending on the size of the organization and the specific time and cost requirements of the project.

A Solution Architect typically straddles the middle position when it comes to Strategy vs Technology focus and Organizational vs Project scope.

The following schematic diagram depicts the different layers in an organization – technology, application, data, people & processes, and business and makes the focus area of the Architect roles very clear.

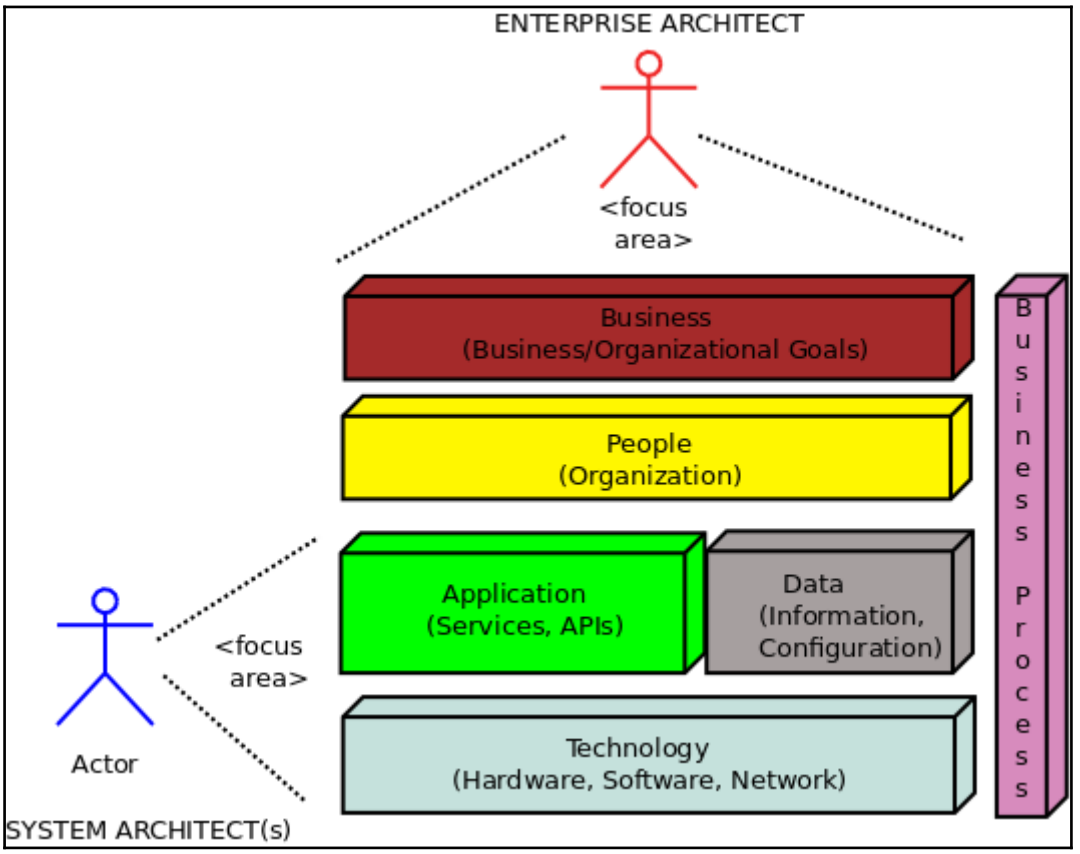

Figure 6: Enterprise vs System Architects

Now that you have understood the picture behind Enterprise and System Architecture, let us get some formal definitions out there.

Quoting from Wikipedia, "**Enterprise architecture (EA)** is a conceptual blueprint that defines the structure and operation of an organization. The intent of enterprise architecture is to determine how an organization can most effectively achieve its current and future objectives."

vs

"A **system architecture** or **systems architecture** is the conceptual model that defines the structural, behavioral, and similar views of a system. A system architecture comprises of systemcomponents, the externally visible properties of those components and the

relationships between them."

It is also important to note that an Enterprise Architect focuses more on the Strategies behind an organization than the details of its systems. An Enterprise Architect need to be a good business and organizational strategist. He may be also a good Technical Architect, at home with technical details, but that would be a bonus, not a requirement for his role. Technical breadth rather than depth is required for the Enterprise Architect's role.

The following diagram depicts the different focus areas and scopes of the different Architect roles we discussed so far.

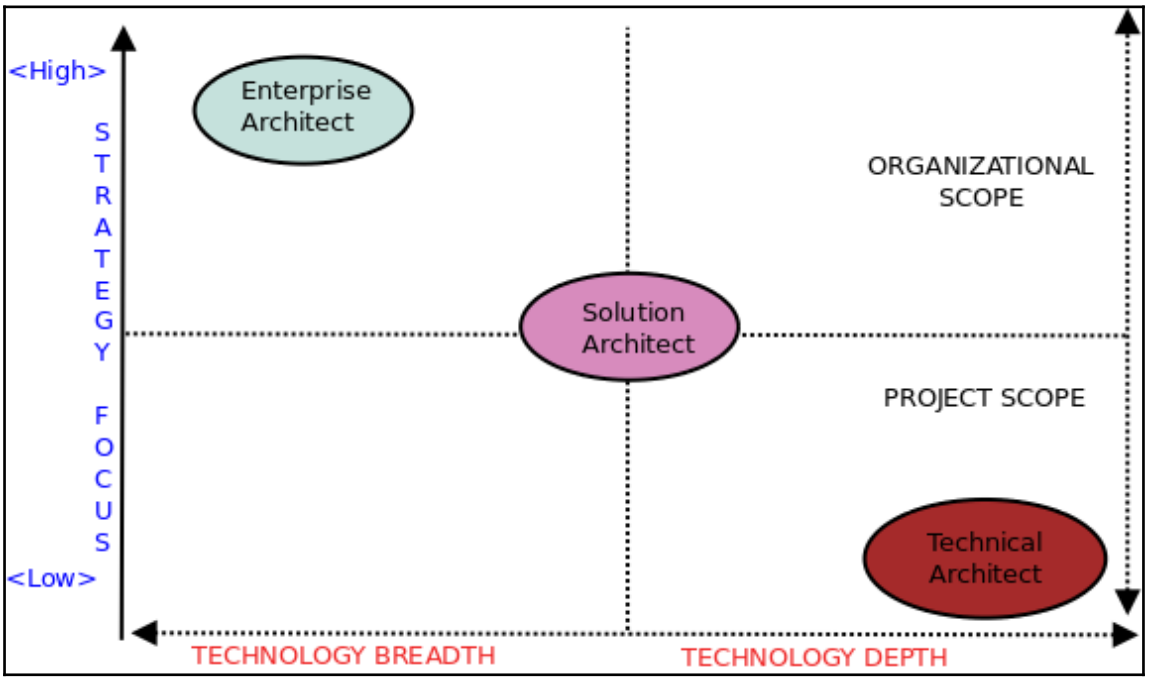

Figure 6: Scope and Focus of various Architect Roles in a Software organization

Architectural Quality Attributes

Let us now discuss an aspect which forms the main topic for the rest of this book – Architectural Quality Attributes.

In a previous section, we discussed how an Architecture balances and optimizes stake

holder requirements. We also saw some examples of contradicting stake holder requirements, which an architect seeks to balance, by choosing an architecture which does the necessary trade-offs.

The term **quality attribute** has been used to loosely define some of these aspects that an architecture makes trade-offs for. It is now the time to formally define what an *Architectural Quality Attribute* is.

"A quality attribute is a measurable,testable property of a system which can be used to evaluate the systems performance within its prescribed environment with respect to its non-functional aspects"

There are a number of aspects that fit this general definition of an Architectural Quality Attribute. However for the rest of this book, we will be focusing on the following quality attributes. These are,

- Modifiability
- Testability
- Scalability & Performance
- Availability
- Security
- Deployability

Modifiability

Many studies show that the about 80% of the cost of a typical software system, occurs after the initial development and deployment. This shows how important modifiability is to a system's initial architecture.

Modifiability can be defined as *the ease at which changes can be done to a system and the flexibility at which the system can adjust to the changes*. It is an important quality attribute as almost every software system changes over its lifetime – to fix issues, for adding new features, for performance improvements etc.

From an architect's perspective, the interest in Modifiability is about,

- Difficulty – Ease at which changes can be made to a system
- Cost – In terms of time and resources required to make the changes
- Risks – Any risk associated with making change to the system

Now, what kind of changes are we talking about here? Is it changes to code, changes to

deployment or changes to the entire architecture? Answer is – it can be at *any* level.

From an architecture perspective, these changes can be captured at generally three levels.

1. **Local** – A local change only affects the specific element. The element can be a piece of code such as a function, a class, a module or a configuration element such as an XML or JSON file. The change *does not cascade* to any neighboring element or to the rest of the system. Local changes are the easiest to make and the least risky of all. The changes can be usually quickly validated with local unit tests.

2. **Non-local –** These changes involve more than one elements. Examples are,

- Modifying a Database Schema which then needs to cascade into the model class representing that schema in the application code
- Adding a new configuration parameter in a JSON file which then needs to be processed by the parser parsing the file and/or the application(s) using the parameter.
- Non-local changes are more difficult to make than local changes and require careful analysis and wherever possible, integration tests to avoid code regressions.

3. **Global –** These changes either involve architectural changes from top-down or changes to elements at the global level that cascade down to a significant part of the software system. Examples are,

- Changing a system's architecture from RestFUL to Messaging (SOAP, XML-RPC etc) based Web services.
- Changing a web application controller from Django to an Angular JS based layer.
- A performance change requirement which needs all data to be pre-loaded at the frontend to avoid any inline model API calls for an online news application.

These changes are the riskiest and also costliest in terms of resources, time and money. An architect needs to carefully vet the different scenarios that may arise from the change and get his team to model them via integration tests. Mocks can be very useful in these kind of large scale changes.

The following table shows the relation between Cost and Risk for the different levels of system modifiability.

Level	**Cost**	**Risk**
Local	Low	Low

Non-local	Medium	Medium
Global	High	High

Table 1: Levels of modifiability associated to Cost & Risk

Modifiability at the code level is also directly related to its Readability.

The more readable a code is the more modifiable it is. Modifiability of a code goes down in proportion to its readability.

The modifiability aspect is also related to the maintainability of the code. A code module which has its element very tightly coupled would yield to modification much lesser than a module which has loosely coupled elementso – this is the *Coupling* aspect of modifiability.

Similarly, a class or module which does not define its role and responsibilities clearly would be more difficult to modify than another one which has well defined responsibility and functionality. This aspect is called *Cohesion* of a software module.

The following table shows the relation between Cohesion & Coupling and modifiability for an imaginary Module "A". Assume that the coupling is from this module to another module "B".

Cohesion	**Coupling**	**Modifiability**
Low	High	Low
Low	Low	Medium
High	High	Medium
High	Low	High

Table 2: Levels of modifiability for an imaginary module "A". The coupling is

with respect to another module "B" (from A->B)

It is pretty clear from the table that having higher Cohesion and lower coupling is the best scenario for the modifiability of a code module.

Other factors that affect modifiability are:

- **Size of a module** (number of lines of code) – Modifiability decreases when size increases.
- **Number of team members working on a module** – Generally, a module becomes less modifiable when a larger number of team members work on the module due to the complexities in merging and maintaining a uniform code base.

- **External third party dependencies of a module –** The larger the number of external third party dependencies, the more difficult the module is to modify. This can be thought of as an extension of the Coupling aspect of a module.
- **Wrong use of the module API –** If there are other modules which make use of the private data of a module rather than (correctly) using its public API, the more difficult it is to modify a module. It is important to ensure proper usage standards of modules in your organization to avoid such scenarios. This can be thought of as an extreme case of tight Coupling.

Testability

Testability refers to how much a software system is amenable to demonstrating its faults through testing. Testability can be thought of as how much a software system *hides* its faults from end users and system integration tests – the more testable a system is the less it is able to hide its faults.

Testability is also related to how *predictable* a software system's behavior is. The more predictable a system, the more it allows for repeatable tests and for developing standard test suites based on a set of input data or criteria. Unpredictable systems are much less amenable to any kind of testing or in the extreme case, not testable at all.

In software testing, you try to control a systems behavior by typically sending it a set of known inputs and then observe the system for a set of known outputs. Both of these combine to form a *testcase.* A *test suite* or *test harness* typically consists of many such test cases.

Test *assertions* are the technique that are used to fail a test case when the output of the element under test does not match the expected output, for the given input. These assertions are usually manually coded at specific steps in the test execution to check the data values at different steps of the testcase.

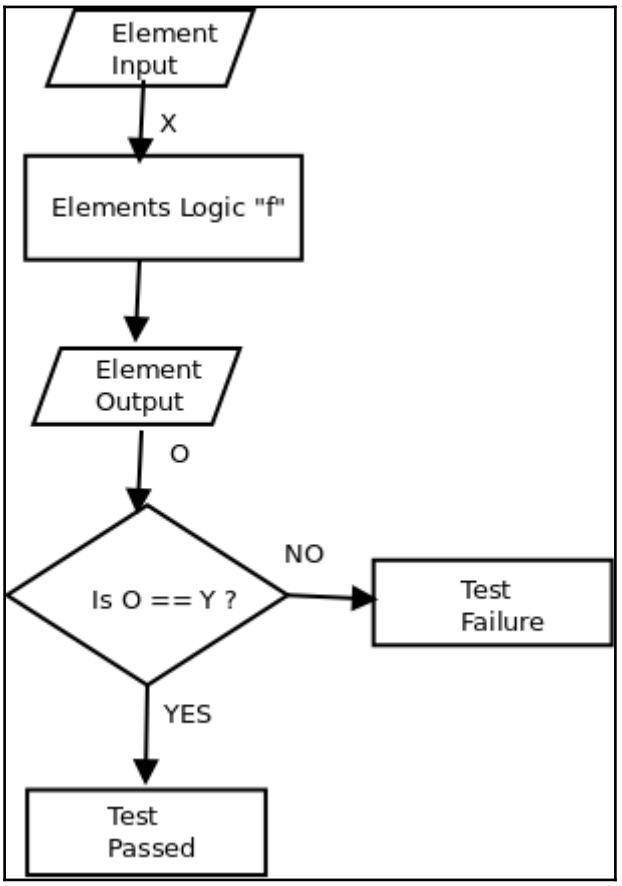

Figure 7: Representative Flowchart of a simple unit test case for function f('X') = 'Y'

The above figure shows an example of a representative flowchart for a testable function "f" for a sample input "X" with expected output "Y". (Of course in a real testcase, assertions instead of a simple check of the output value is used.)

In order to recreate the session or state at the time of a failure, the *record/playback* strategy is often used. This employs specialized software (such as Selenium) which records all user actions which led to a specific fault and saves it as a test-case. The test is reproduced by *replaying* the testcase using the same software which tries to simulate the same testcase – by repeating the same set and order of UI actions. Record/playback strategy is used typically for web applications.

Testability is also related to the complexity of code in a way very similar to modifiability. A

system becomes more testable when parts of it can be isolated and made to work independent of the rest of the system. In other words, a system with *Low Coupling* is more testable than a system with *High Coupling*. Note how similar this behaves to the Modifiability aspect. Similar is the case with *Cohesion* as well.

Another aspect of testability, which is related to the *predictability* mentioned above, is to reduce *non-determinism*. When writing test suites, we need to isolate the elements that are to be tested from other parts of the system that have a tendency to behave unpredictably so that the tested elements behavior becomes predictable.

An example is a multi-threaded system which responds to events raised in other parts of the system. The entire system is quite possibly unpredictable and not amenable to repeated testing – instead one needs to separate the *events* sub-system and possibly *mock* its behavior – so that those inputs can be controlled and the sub-system which receives the events become predictable and hence testable.

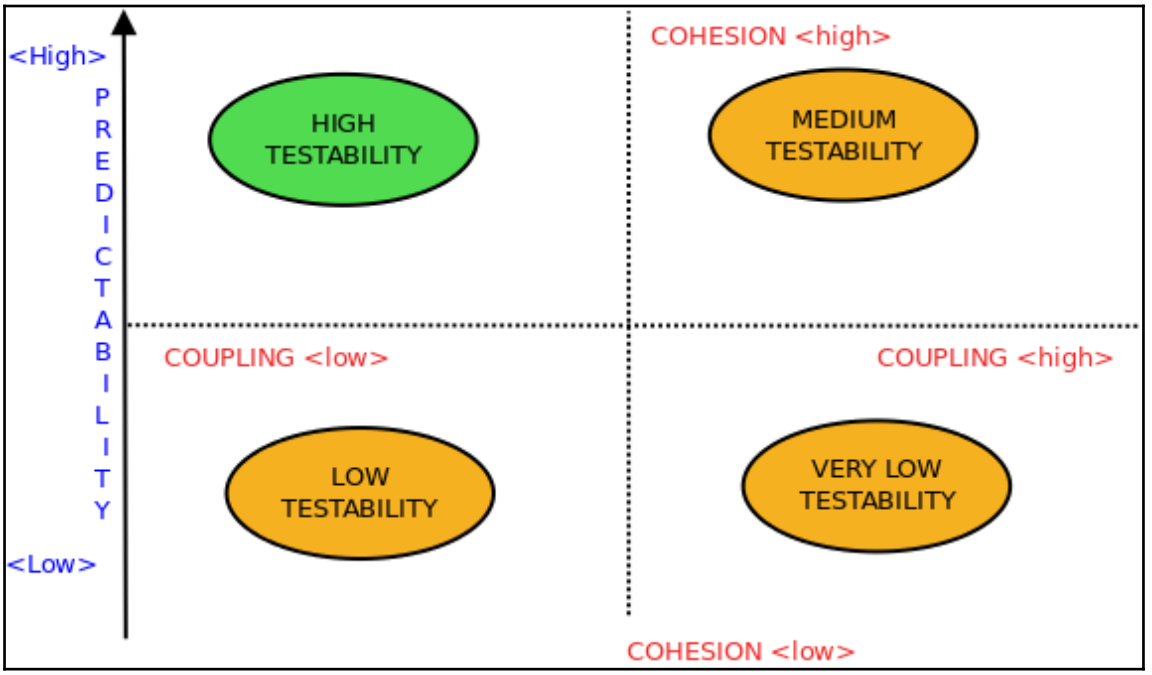

Figure 7: Relation of Testability to Predictability of a system to Coupling & Cohesion

Scalability/Performance

Modern day web applications seem all about "scaling up". If you are part of any modern day software organization, it is a almost a rule that you have heard about or worked on an application that is written "for the cloud", which is able to scale up elastically on demand.

"Scalability is the capability of a system to accommodate an expanding workload on demand. A system that is architected to scale will be able to increase its level of performance elastically when larger operational demands are applied to it."

Scalability in the context of a software system, typically falls into two categories.

- **Horizontal Scalability** – Horizontal scalability implies *scaling out/in* a software system by adding more computing nodes to it. Advances in cluster computing in the last decade has given rise to the advent of commercial horizontally scalable *elastic* systems as services on the Web. A well-known example is Amazon Web Services.

  In horizontally scalable systems, typically data and/or computation is done on units or nodes – which are usually virtual machines running on commodity systems known as virtual private servers (VPS). The scalability is achieved "n" times by adding "n" or more nodes to the system, typically fronted by a load balancer. Practically, the scalability factor is never perfectly linear beyond a certain number of systems however.

  Scaling out means expanding the scalability by adding more nodes and *scaling in* means reducing the scalability by removing existing nodes.

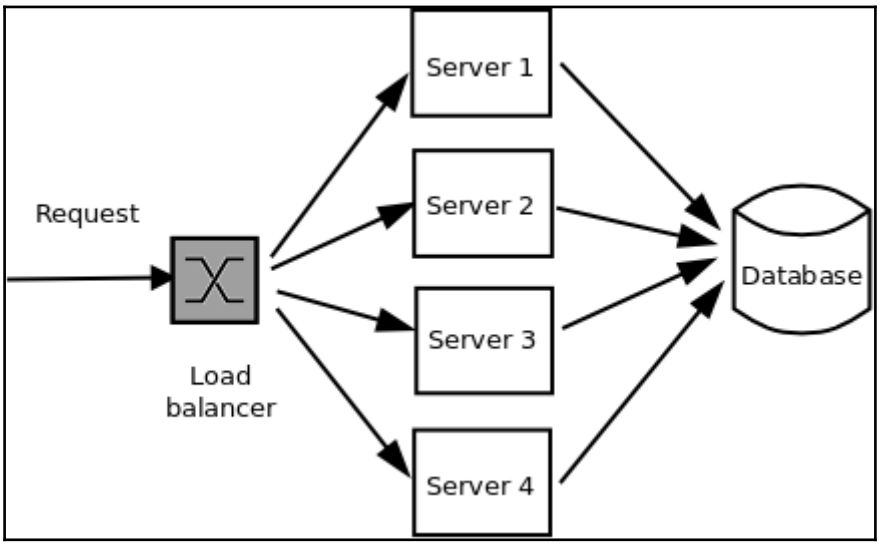

Figure 8: Example deployment architecture showing Horizontally Scaling a web application server

- **Vertical Scalability** – Vertical Scalability involves adding or remove resources from a single node in a system. This is usually done by adding or removing CPUs or RAM (memory) from a single server in a cluster. The former is called *scaling up* and latter, *scaling down.*

  Another kind of *scaling up* is increasing the capacity of an existing software process in the system – typically by supplementing its computation power – this is usually done by increasing the number of processes or threads available to an application.

  Some examples are:

- Increasing the capacity of an Nginx server process by increasing its number of worker processes
- Increasing the capacity of a PostgreSQL server by increasing its number of maximum connections.

Performance of a system is related to its scalability. Performance of a system can be defined as:

Performance of a computer system is the amount of work accomplished by a system using a given

unit of computing resource. Higher the work/unit ratio, higher the performance.

The unit of computing resource to measure performance can be one of

- Response Time – How much time a function or any unit of execution takes to execute in terms of real time (user time) and clock time (CPU time).
- Latency – How much time it takes for a system to get its stimulation and then provide a response. An example is the time it takes for the request-response loop of a web application to complete, measured from the end-user perspective.
- Throughput – The rate at which a system processes its information. A system which has higher performance would usually have a higher throughput and correspondingly higher scalability. An example is the throughput of an E-commerce website measured as the number of transactions completed per minute.
- Processing Speed – This is directly related to the type and capacity – measured in terms of raw frequency (GHz) and number of cores – of the CPU the system is running on and the kind of workloads the system produce. For example an application that does parallel computing on multiple threads would have better performance on a multi-core (SMP) CPU than on a single core CPU.

Performance is closely tied to scalability, especially vertical scalability. A system that has excellent performance with respect to its memory usage would easily scale up vertically by adding more RAM.

Similarly a system that has multi-threaded workload characteristics and is written optimally for a multi-core CPU, would scale up by adding more CPU cores.

Horizontal scalability is thought of as having no direct connection to the performance of a system within its own compute node. However if a system is written in a way that it doesn't utilize the network effectively, thereby producing network latency issues, it may have a problem scaling horizontally effectively as the time spent on network latency would offset any gain in scalability obtained by distributing the work.

Some dynamic programming languages like Python has built-in scalability issues when it comes to vertically scaling up . For example the Global Interpreter Lock (GIL) of Python (CPython) prevents it from making full use of the available CPU cores for computing by multiple threads.

Availability

Availability refers to the property of *readiness* of a software system to carry out its operations when the need arises.

Availability of a system is closely related to its *reliability.* The more reliable a system is the more available it becomes.

Another factor which modifies availability is the ability of a system to recover from faults. A system may be very reliable, but if the system is unable to recover either from complete or partial failures of its sub-systems, then it may not be able to guarantee availability. This aspect is called *recovery*.

Hence a more technical definition of availability is,

Availability of system is the degree or proportion at which a system is in a fully operable state, when the system is called for a mission at an unknown, random time.

Mathematically this can be expressed as:

Availability = MTBF/(MTBF + MTTR)

where,

- MTBF – Mean time between failures
- MTTR – Mean time to repair

This is often called as the *mission capable rate* of a system.

Techniques for Availability are closely tied to recovery techniques. This is due to the fact that a system can never be 100% available. Instead one needs to plan for faults and strategies to recover from faults which directly determine the availability. These techniques can be classified as,

- **Fault Detection** – Ability to detect faults and take action helps to avert situations where a system or parts of a system become unavailable completely. Fault detection typically involves steps like Monitoring, Heartbeat & Ping/Echo messages which are sent to the nodes in a system and response measured to calculate if the nodes are alive, dead or in the process of failing.
- **Fault Recovery** – Once fault is detected, the next step is to prepare the system to recover from the fault and bring it to a state where the system can be considered available. Typical tactics used here include Hot/Warm Spares (Active/Passive redundancy), Rollback, Graceful Degradation and Retry.
- **Fault Prevention** – This approach uses active methods to anticipate and prevent

faults from occurring so that the system does not have a chance to go to recovery.

Availability of a system is closely tied to the Consistency of its data via the CAP theorem. We will look at this in detail in the Chapter on Availability.

Availability is also tied to the system's Scalability tactics, Performance metrics and its Security. For example, a system that is highly horizontally scalable would have a very high availability since it allows the load-balancer to determine inactive nodes and take them out of the configuration pretty quickly.

A system which instead tries to *scale up* may have to monitor its Performance metrics carefully. Even when the node on which the system is fully available, the system may have availability issues if the software processes are squeezed for system resources – such as CPU time or Memory. This is where Performance measurements become critical and the system's load factor needs to be monitored and optimized.

With the increasing popularity of Web applications and distributed computing, security is also an aspect that affects availability. It is possible for a malicious hacker to launch remote Denial of Service attacks on your servers – and if the system is not foolproofed against such attacks – can lead to a condition where the system becomes unavailable or partially available.

Security

Security, in the software domain can be defined as the degree of ability of a system to avoid harm to its data and logic from unauthenticated access, while continuing to provide services to other systems and roles that are properly authenticated.

A security crisis or attack occurs when a system is intentionally compromised with a view to gaining illegal access to it in order to compromise its services, copy or modify its data or deny access to its legitimate users.

In modern software systems, the users are tied to specific roles which have exclusive rights to different parts of the system. For example a typical web application with a database may define the following roles.

- user – End user of the system with login and access to his/her private data.
- dbadmin – Database administrator role who can view/modify or delete all database data.
- reports – Report admin role who has admin rights only to parts of database and code that deal with report generation.

- admin – Super-user role who has edit rights to the complete system.

This way of allocating system control via user roles is called *access control*. Access control works by associating a user role with certain *system privileges* thereby decoupling the actual user login from the rights granted by these privileges.

This principle is the *Authorization* technique of Security.

Another aspect of security is with respect to transactions where each person must validate the actual identity of the other. Public key cryptography, message signing etc are common techniques used here. For example when you sign an email with your GPG or PGP key, you are validating yourself – The sender of this message is really me – Mr. A to your friend Mr. B on the other side of the email. This principle is the *Authentication* technique of Security.

Other aspects of Security are:

- **Integrity** – Techniques used to ensure that a data or information is not tampered with in anyway on its way to the end user. Examples are Message Hashing, CRC Checksum etc.
- **Origin** – Techniques used to ensure the end receiver that the origin of the data is exactly the same as where it is purporting to be from. Examples of this are SPF, Sender-ID (for email), Public Key Certificates & Chains (for websites using SSL) etc.
- **Authenticity** – Techniques which combine both Integrity and Origin of a message into one. This ensures that the author of a message cannot deny the contents of the message as well as its origin (himself/herself). This typically uses Digital Certificate Mechanisms.

Deployability

Deployability is one of those quality attributes which is not fundamental to the software. However, in this book we are interested in this aspect because it plays a critical role in many aspects of the ecosystem in the Python programming language and its usefulness.

Deployability is the degree of ease at which software can be deployed from the development to the production environment. It is more of a function of the technical environment, module structures and programming runtime/languages used in building a system and has nothing to do with the actual logic or code of the system.

Some factors that determine deployability are,

- **Module structures** – If your system has its code organized into well-defined

modules/projects which compartmentalize the system into easily deployable sub-units, the deployment is much easier. On the other hand if the code is organized into a monolithic project with a single setup step, it would be hard to deploy the code into a distributed, horizontally scalable cluster.

- **Production vs Development Environment** – Having a production environment which is very similar to the structure of the development environment, makes deployment an easy task. When the environments are similar, the same set of scripts and tool-chains that are used by the developers/Devops team can be used to deploy the system to a development server as well as a production server with minor changes – mostly in the configuration.
- **Development Ecosystem Support** – Having a mature tool-chain support for your system runtime which allows configurations such as dependencies to be automatically established and satisfied, increases deployability. Programming languages such as Python are rich in this kind of support in its development ecosystem with a rich array of tools available for the Devops professional to take advantage of.
- **Standardized Configuration** – It is a good idea to keep your configuration structures (files, database tables etc) the same for both developer and production environments. The actual objects or filenames can be different, but if the configuration structures vary widely across both the environments, deployability decreases as extra work is required in order to map the configuration of the environment to its structures as one goes from one environment to another.
- **Standardized Infrastructure** – It is a well-known fact that keeping your deployments to a homogeneous or standardized set of infrastructure greatly aids deployability. For example, if you standardize your Frontend application to run on 4G RAM, Debian-based 64-bit Linux VPS, then it is easy to automate deployment of such nodes – either using a script or by using elastic compute approaches of providers like Amazon – and to keep a standard set of scripts across both development and production environments.

On the other hand if your production deployment consists of heterogeneous infrastructure – say a mix of Windows and Linux servers with varying capacities and resource specifications – the work typically doubles for each type of infrastructure – decreasing deployability.

- **Use of Containers** – The user of container software, popularized by the advent of technology like Docker and Vagrant built on top of Linux containers, has become a recent trend in Deploying software on servers. The use of containers allows to standardize your software and makes deployability easier by reducing the amount of overhead required to start/stop the nodes as containers don't come with the overhead of a full virtual machine. This is an interesting trend to watch for.

Summary

In this chapter, we learned about Software Architecture. We saw the different aspects of software architecture and learned that every architecture comprises a system which has structure working in an environment for its stakeholders. We briefly looked at how Software Architecture differs from Software Design.

We went on to look at various characteristics of Software Architecture – such as how a software architecture defines a structure, picks a core set of elements and connects stakeholders.

We then addressed the important question of the importance of Software Architecture to an organization and why it is a good idea to have a formal software architecture defined for your software systems.

The distinction of different roles of Architects in an organization was discussed next. We saw the various roles System Architects play in an organization and how Enterprise Architect's focus is different from that of the System Architects. The focus of strategy and technology breadth vs technology depth was clarified with illustrations.

We then learned the elements of the main theme of this book – Architectural Quality Attributes. We defined what a quality attribute is and then looked in quite some detail on the quality attributes of Modifiability, Testability, Scalability/Performance, Security & Deployability. While learning the details of these attributes we discussed their definitions, techniques and how they often relate to each other.

With the preparation from this chapter, we are now ready to take on these quality attributes and then discuss it at detail on the various tactics and techniques to achieve them using the Python programming language. That forms the rest of this book.