

# Software Architecture Document Guidelines

The purpose of the software architecture document (SAD) is to provide the following :

1. An outline description of the software architecture, including major software components and their interactions.
2. A common understanding of the architectural principles used during design and implementation.
3. A description of the hardware and software platforms on which the system is built and deployed.
4. Explicit justification of how the architecture meets the non-functional requirements.

The following sections illustrate the content that should be included in a software architecture document, but it may not be mandatory for all software projects when taking into account factors such as project size, etc.

If the information is available elsewhere, the software architecture document should make a reference to that source rather than repeating it (e.g. non-functional requirements may be stated in the requirements documentation and designs may be in separate design documents or a UML model).

The SAD is a living document and may not contain all of the information listed here at any point in time. If this is the case, this document can be used to provide guidance as to the sort of additional information that should be captured.

## 1. Functional View

The functional view allows you to summarise what the key, and architecturally significant, functions of the system are. It allows you to make an explicit link between the functional aspects of the system (use cases, stories, etc) and explain why they are significant to your architecture.

### Checklist

- Is it clear which features/functions/use cases are significant to the architecture?
- It is clear that these have been used to shape and define the architecture?

## 2. Non-functional View

The non-functional view allows you to re-iterate or summarise the key non-functional requirements and, again, explicitly highlight those that are deemed as architecturally significant. Each non-functional requirement should be precise, leaving no interpretation to the reader. Examples where this isn't the case include :

- “the request must be serviced quickly”
- “there should be no overhead”
- “as fast as possible”
- “as small as possible”
- “as many clients as possible”

Example non-functional requirements include the following :

- Performance (e.g. latency and throughput)

- Scalability (e.g. data and traffic volumes)
- Security (e.g. authentication, authorisation, data confidentiality)
- Extensibility
- Flexibility
- Auditability
- Monitoring and management
- Reliability
- Availability
- Failover/disaster recovery targets (e.g. manual vs automatic, how long will this take?)
- Interoperability
- Legal and regulatory requirements (e.g. data protection act)
- Internationalisation and localisation

### Checklist

- Is there a clear understanding of the non-functional requirements that the architecture must satisfy?
- Are the non-functional requirements quantifiable and testable?
- Have common non-functional requirements been explicitly marked as out of scope if they are not needed (e.g. “user interface elements will only be presented in English”).
- Are any of the non-functional requirements unrealistic? (e.g. true 24x7 availability is typically very costly to implement).

## 3. Architectural Principles

The architectural principles section allows you to highlight those principles that have been used in defining the architecture. These could have explicitly asked for or they can be principles that \*you\* want to follow. Examples might include the following :

- The use of common design patterns or patterns established by other projects.
- The use of a rules engine for business rules.
- Composition over inheritance.
- The ability to start and stop components at runtime.
- The ability to effect new configuration changes across all components at runtime.

### Checklist

- Are there any other principles (e.g. other non-functional requirements that have not been explicitly requested) that have helped influence the architecture?

## 4. Architectural Constraints

Software lives within the context of the real-world, and the real-world has constraints. This section allows you to state these constraints so that it is clear that you are working within them and clear how they affect your architecture decisions. Example constraints include :

- Approved technology lists and technology constraints.
- Local standards (e.g. development, coding, etc).
- Public standards (e.g. HTTP, SOAP, XML, XML Schema, WSDL, etc).
- Standard protocols.
- Standard message formats.
- Skill profile of the development team.

- Project deadlines.
- Nature of the project (e.g. tactical or strategic).

### Checklist

- Are the constraints well documented and comprehensive?
- Is it clear how the constraints affect the architecture?

## 5. Process View

The process view allows you to show *what* the system is doing at a high level, and it also provides you with an opportunity to show how the smaller steps within the process fit together. This doesn't have to be a long section, but it's important that aspects such as parallelism are represented here, where processes fork or join, etc.

### Checklist

- Is it clear what the system does from a process perspective?
- Are the major flows of information through the system well understood and documented (e.g. using UML activity diagrams)?

## 6. Logical View

The logical view allows you to present the structure of the system through its components and their interactions. Typically this will show high level technology choices.

### Checklist

- Is a logical view of the architecture clearly portrayed?
- Does it show the major components and interfaces?
- Are they described at a high level?
- Does the logical view show external systems and any other dependencies at a high level (low level detail about the dependencies isn't required here)?

## 7. Interface View

Closely related to the logical view is the interface view. Interfaces are one of the riskiest parts of any software system, so it's very useful to show what the internal/external interfaces are and how they work.

### Checklist

- Are the key internal (e.g. databases, messaging systems, etc) and external interfaces (e.g. other systems) well specified at a high level?
- If messaging is being used, which queues and topics are components using to communicate?
- What format are the messages (e.g. plain text or XML defined by a DTD/Schema)?
- Are they synchronous or asynchronous?
- Are asynchronous messaging links guaranteed?
- Are subscribers durable where necessary?
- Can messages be received out of order and is this a problem?
- Who has ownership of the interfaces?

## 8. Technology Selection

Including a technology selection section in your software architecture document gives you somewhere to document the decisions that went into choosing or not

choosing technologies. It's often useful to have a summary of these decisions in an appendix or separate document so that they can be referred to later in the project.

### Checklist

- Is it clear why the selected technologies were chosen?
- If there were options, why were they not chosen?
- Do they all fit in with the constraints outlined previously?
- Are all software and hardware tiers covered?

## 9. Design View

The design view is where the lower level implementation details start to make an appearance. For example, this could include information such as how your architectural layering will be implemented through to documenting blueprints/ common usage patterns for the technologies/frameworks you have chosen for the implementation.

### Checklist

- Is it well understood how the key use cases will be implemented?
- How are the chosen technologies used and combined?
- Are there common patterns across the architecture?
- If yes, are these well understood and documented?
- Are the diagrams (e.g. UML class and sequence) up to date and do they reflect reality?
- Are any common wheels being reinvented? If so, why aren't vendor/open source products being used?
- Is there enough information here to provide the rest of the development team with an overview/the intent of how the designs work?

## 10. Infrastructure View

The infrastructure view is used to describe the physical hardware and networks on which the software will be deployed.

### Checklist

- Is there a clear physical architecture?
- What hardware does this include across all tiers?
- Does it cater for redundancy, failover and disaster recovery if applicable?
- Is it clear how the chosen hardware components have been sized?
- If multiple boxes and sites are used, what are the network links between them?
- Who is responsible for support and maintenance of the infrastructure?
- Are there central teams to look after common infrastructure (e.g. databases, message buses, application servers, networks, routers, switches, load balancers, reverse proxies, internet connections, etc)?
- Who owns the resources?
- Are there sufficient resources for development, testing, acceptance, pre-production, production, etc?

## 11. Deployment View

The deployment view details *how* the software will be deployed onto the physical infrastructure.

## Checklist

- Is it clear how the software components will be deployed across the hardware elements described in the physical view?
- If this is still to be decided, what are the options and have they been documented?
- Is it understood how memory and CPU will be partitioned between the processes running on a single hardware node?
- Which components are active-active and which are active-passive?
- Is it clear how data is replicated across sites?
- Has the rollout and recovery strategy been defined (this might be in a separate document, but referenced)?

## 12. Security View

Security is an important aspect of most systems, so it's essential that it is thought about and documented clearly. Having a dedicated security section provides a way to explain how your architecture will meet security requirements such as authentication, authorisation, data confidentiality, etc. Some organisations have specialised security teams that will help with and/or need to review your work in this area before deploying your system into a production environment. Being explicit about security helps you spot any holes before it's too late.

### Checklist

Is there a clear understanding of how security is handled within the architecture and how any security requirements have been satisfied? This includes the following :

- Authentication.
- Authorisation.
- Confidentiality of data between components (e.g. during user login, during requests between components, using technologies such as web services or messaging, across public networks).
- Non-repudiation.
- Different types of users and their roles.
- The use of a security realm or integration with in-house single sign-on mechanisms.
- Network separation using firewalls and DMZs (red, amber, green model).
- Restricted access to resources.
- Permissioning of data on a per user/role/etc basis and the ability to modify those permissions.
- Storage of credentials (e.g. database logons).
- Distribution of certificates and keys.

## 13. Monitoring, Management and Administration view

Most systems will be subject to support and operational requirements, particularly around how they are monitored, managed and administered. Including a dedicated section in the software architecture document again lets you be explicit about how your architecture will support those requirements, particularly if you enlist the help of the team that are ultimately going to support your system.

### Checklist

- Is it clear how the architecture provides the ability for operation/support teams to monitor and manage the system?

- How is this achieved across all tiers of the architecture (e.g. from client tier to database)?

## 14. Data View

A specific data view is worth including in your architecture diagram if your system is data-centric; managing a large quantity of data or dealing with complex data flows. This section can include information such as sizing and capacity planning through to archives and backups.

### Checklist

- Is there a high level understanding of how much storage will be required to persist data?
- What are the archiving strategies?
- Are there any regulatory requirements for the long term archival of business data?
- Likewise for log files and audit trails?

## 15. Justification of the Non-functional Requirements

This section provides a way for you to explicitly state how the non-functional requirements you stated in the non-functional view are satisfied by your architecture. This is an important yet often forgotten piece of the architecture puzzle, and clearly communicating your architecture's fitness for purpose will help provide everybody with the confidence that your solution will work. If you built an executable reference architecture to prove some of the key non-functional requirements, this is where you can reference that work and the proof of its success.

### Checklist

- For each of the non-functional requirements, is it explicit how the architecture satisfies it?
- In the case of performance and scalability targets, are the test cases and results referenced?
- If true 24x7 availability is required, is redundancy and automatic failover built into all aspects of the architecture?
- Are there any single points of failure?
- What happens if a component fails?
- What happens if an external system you rely on fails?
- Does this affect your availability? Are your transactions ACID?
- Would you have 2PC transactions in-doubt?
- Can you recover from a system failure?
- Who is responsible for system recovery and failover?
- Can you recover in a business continuity scenario?
- Will all data have been replicated between sites?
- How do you tell system components to use alternative resources in the event of DR/BCP?
- Has the architecture been reviewed by in-house security risk teams (if applicable)?