SWATANTRA.ai

SYSTEM DESIGN

OVERVIEW OF THE ART OF SCALABLE AND RELIABLE SYSTEMS

A COMPLETE FOUR-VOLUME CURRICULUM (VOL. 1 OF 4)

By Sharath and Tharun

A GUIDE FOR SOFTWARE ARCHITECTS AND DEVELOPERS

Table Of Contents

PREFACE TO PART I

Foundations of System Design

A first release by Sharath and Tharun B S

This book is our first release, and it represents our decision to turn practical engineering experience, hard-won lessons, and continuous learning into a structured guide for people who want to design systems with clarity rather than guesswork.

Part I is deliberately about foundations. Before choosing databases, queues, caches, cloud services, or distributed-system patterns, a designer must understand the problem being solved, the requirements that matter, and the consequences those requirements create for architecture. Technology is important, but it comes after the shape of the problem is understood.

The five chapters therefore follow a practical sequence: what system design is; how requirements are gathered; how functional and non-functional requirements differ; how those requirements become a high-level architecture; and how that architecture is translated into low-level, implementable detail. Throughout the book, the same concerns keep returning - scalability, reliability, availability, security, data ownership, integration, observability, operations, and trade-offs.

This volume does not ask you to memorize fashionable patterns. It asks you to explain why a component exists, what requirement justifies it, what can fail, what grows, what must be protected, and what new complexity a design choice introduces. That reasoning is the foundation on which the later volumes can go deeper into scalable and distributed systems.

We hope this book becomes a practical companion for students, engineers, technical leads, architects, and business technologists who want to move beyond surface-level diagrams and develop a repeatable way to think about software architecture.

Welcome to Part I. Start with the problem, then earn the architecture.

Chapter Roadmap

Part I is organized as a five-chapter progression. Each chapter adds one layer of design reasoning, moving from the problem and requirements to architecture and implementation detail. Read in order if system design is new to you; use the roadmap as a reference if you already know the basics.

Chapter 1: Introduction to System Design

Explains what system design is, why it matters, and how architecture connects business goals to working software. It introduces scalability, reliability, availability, security, data architecture, integration, cloud-native design, observability, and trade-off thinking.

Chapter 2: Requirements Gathering

Shows how to discover, validate, and document business, functional, non-functional, security, operational, and integration requirements before selecting architecture or technology.

Chapter 3: Functional vs Non-Functional Requirements

Separates what the system must do from how well it must work, and shows how quality attributes such as performance, security, reliability, scalability, maintainability, and operability shape technical design.

Chapter 4: High-Level Architecture

Builds the broad blueprint: major components, responsibilities, system boundaries, data ownership, integrations, deployment concerns, security, scalability, reliability, operations, and architectural decisions.

Chapter 5: Low-Level Design

Turns the architectural blueprint into implementable detail: classes, APIs, schemas, validation rules, error handling, data access, design patterns, testing, performance, security, and operational hooks.

The purpose of this sequence is simple: do not jump from a business request directly to code or technology. First understand the need, then make the architecture visible, and only then design the implementation detail.

How to Read This Book

System design is not a memorized list of technologies. It is the habit of connecting business goals, user behavior, data, failure modes, security, operations, cost, and future change into one coherent solution. Read the book with that habit in mind.

1. If system design is new to you, read Chapters 1 through 5 in order. Each chapter depends on questions introduced earlier.
2. For every component or pattern, ask which requirement justifies it. If you cannot answer that, the component may be unnecessary complexity.
3. When an example introduces a design choice, identify both the benefit and the cost. More scalability, flexibility, security, or availability usually introduces new complexity somewhere else.
4. Do not treat functional requirements as the whole problem. Performance, security, reliability, auditability, operability, and maintainability can change the architecture just as much as visible features.
5. Keep High-Level Design and Low-Level Design aligned. A sound architecture with weak implementation detail still fails; excellent code inside a weak architecture still becomes difficult to scale and operate.
6. Use the Architecture Review Checklist after each design. A diagram is not complete until failure, ownership, security, operations, cost, and change have been considered.

How This Book Benefits You

By the end of Part I, you should be better able to:

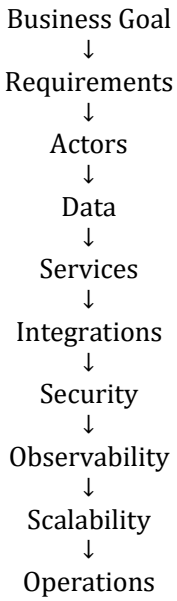
- translate a business goal into clear functional and non-functional requirements;
- identify the actors, data, boundaries, integrations, and quality attributes that shape a system;
- design and explain a high-level architecture instead of jumping straight to technology;
- turn that architecture into APIs, classes, schemas, validation, error handling, and testable low-level design;
- compare design alternatives through trade-offs rather than popularity;
- review a design for scalability, reliability, security, data ownership, observability, operations, and future change.

How This Book Works

The book repeatedly uses the same reasoning path. The examples change, but the order of thinking should remain stable. That order is more important than memorizing a specific stack.

1. **Start with the business goal:** What outcome is the system supposed to create or improve?
2. **Make the requirements explicit:** What must the system do, and how well must it work?
3. **Identify actors, data, and boundaries:** Who uses or operates the system, what information matters, who owns it, and where does responsibility change?
4. **Shape the high-level architecture:** Separate major responsibilities, integrations, data stores, communication paths, security boundaries, and deployment concerns.
5. **Translate the architecture into low-level design:** Define APIs, classes, schemas, rules, errors, persistence behavior, testing, and operational details.
6. **Review the trade-offs:** Ask what fails, what scales, what costs grow, what must be protected, and what complexity the chosen design introduces.

System Design Thinking Framework



A beginner often starts with technology: “Should I use Kafka, Redis, Kubernetes, or microservices?” A stronger designer starts with the business goal and works downward. Technology comes after the shape of the problem is understood.

Architecture Review Checklist

Use this checklist whenever you review a design. A design that looks good on a diagram may still fail in production if these questions are ignored.

1. Can the system scale in the dimension that actually matters: users, data volume, requests, geography, tenants, or workflow complexity?
2. What happens when a database, queue, cache, third-party API, or downstream service fails?
3. Where is the source of truth for each important business object?
4. What are the likely bottlenecks under peak load?
5. How is data secured in transit, at rest, in logs, in backups, and in analytics systems?
6. How are authentication, authorization, and least-privilege access enforced?
7. How will the team debug slow requests or failed business workflows?
8. What are the expected operational costs, and which costs grow with traffic?
9. How are schema changes, API changes, and future features accommodated?
10. What decisions are documented, and what alternatives were rejected?

A useful review does not ask whether the architecture uses enough modern technology. It asks whether the design can be justified, operated, protected, changed, and recovered when reality is less clean than the diagram.

About the Authors

Sharath

Principal Engineer | Distributed Data Infrastructure | AI Architect at stealth startup

Sharath is a Principal Engineer with twenty years of experience designing distributed data infrastructure for Fortune 500 technology companies. His work focuses on high-throughput messaging pipelines, distributed data systems, and Agentic AI. He brings a production-first perspective shaped by systems that have been designed, stressed, broken, recovered, and improved at scale.

Tharun B S

Junior AI/ML Engineer

Tharun B S is a junior AI/ML Engineer with a strong interest in agentic AI, fault-tolerant system design, and the connection between technical architecture and business strategy. He contributes a builder's perspective focused on making complex system-design concepts understandable, practical, and useful for emerging engineers.

Acknowledgement

We sincerely thank Kalyan for reviewing this book. Kalyan's feedback helped us refine the content, improve clarity, and strengthen the quality of this first release.

— Sharath and Tharun B S

Before You Begin Part I

Keep these six rules in mind as you move into Chapter 1:

- Start with the problem and requirements, not with a preferred technology.
- Treat functional and non-functional requirements as equally important architecture inputs.
- Ask what fails, what scales, what must be protected, and how the system will be operated.
- Use patterns only when the problem they solve is important enough to justify their complexity.
- Keep High-Level Design and Low-Level Design aligned as the design evolves.
- Document important decisions and the trade-offs behind them.

Next: Chapter 1 — Introduction to System Design

Chapter 1 : Introduction to System Design

This chapter covers the concepts, patterns, trade-offs, architectural decisions and implementation concerns related to introduction to system design.

It includes design considerations, scalability implications, operational concerns, security considerations and areas for future expansion.

1.1 What is System Design?

System design is really about figuring out how all the pieces of a software system connect so the end product actually delivers what a business needs —and does it reliably. You're looking at everything here: the overall architecture, how different services work together, the way data moves around, APIs, how you handle storage and security, making sure the system can scale, and more. Basically, it's the layer between the business goal and the code developers end up writing.

Think about something like a payment service. If you start simple, you probably process every payment right when the user clicks "Pay." The system talks to the payment provider, waits for an answer, updates the order, and tells the user what happened. It's straightforward and easy to follow. But once you get more users, this setup starts to crack. Every time a payment provider takes too long, your system is stuck waiting—and so is your user.

Switching to an asynchronous setup helps. You insert payment requests onto a queue, process them in the background, and your system isn't tied up waiting for outside responses. That makes it easier to handle traffic spikes. But this introduces new headaches: now you have to keep track of what's still in progress, deal with failed payments and retries, watch out for duplicate callbacks, sort out reconciliation, and decide how to keep users informed.

There's no single right answer. The best approach comes down to what your users expect, how much risk the business can handle, how reliable your partners are, how much traffic you think you'll get, and how well you can operate the thing. That's what system design really is—it's not about following trends, it's about picking the approach that actually works for the problem in front of you.

1.2 Why System Design Matters

Good system design is really the difference between software that's a joy to work with and software that's a pain. If the design's solid, you can change things easily, avoid nasty bugs, keep everything running safely, and actually figure out what's wrong when things break. Weak design?

Well, it might get through a demo. Maybe it even holds up for the first few users. But the real problems show up later—when more people join, the data gets messy, integrations sputter, security audits start, or some new folks have to maintain the whole thing.

Take a chat app, for example. Stashing all messages in a single database table works fine at first. But as you grow, the database starts to choke—writes and reads slow down, indexes struggle, and storage balloons.

The team might need to split things up (partitioning), archive old messages, add caching, spread messages out with fanout strategies, use queues for delivery, or even separate how the app reads messages from how it writes them. Each fix solves a problem, sure, but also adds complexity. It's smart to keep things simple until you really need more complexity—don't just add layers because they look cool.

Design touches security, too. Leave authentication, authorization, or audit until the end and you'll probably be tossing out or rewriting big chunks later. If nobody owns the data, duplicates pile up and people fight over which version is the real one. And if there's no way to see what's happening in production, figuring out issues is basically a shot in the dark.

1.3 Functional and Non-Functional Requirements

Functional requirements spell out what the system actually does. Stuff like—users can sign up, managers approve leave, the system creates invoices, or admins export reports.

On the other hand, non-functional requirements focus on how the system handles all that work. Think of things like speed, reliability, security, how easy it is to keep the system running, user-friendliness, or whether it meets legal standards.

Take these examples: A functional requirement looks like, "Users can submit reimbursement claims."

Non-functional? That would be, "The reimbursement submission API answers within two seconds for 95% of requests during normal traffic." That second one packs a punch. It doesn't just affect how code is written, but it also changes the database setup, how you validate inputs, deal with file uploads, manage queues—basically, it shapes the whole system behind the scenes.

1.4 High-Level Design

High-level design describes the broad structure of the system: major components, responsibilities, integrations, data stores, communication paths, and deployment approach. Low-level design translates that broad structure into implementable details: APIs, classes, schemas, validation rules, error handling, sequence flows, and internal logic.

Both levels matter. A good high-level design with poor low-level design becomes difficult to implement. Good low-level code inside a weak architecture becomes difficult to scale and operate. System design joins both views.

1.5 Architecture Components

Most systems include familiar building blocks: user interfaces, APIs, application services, databases, caches, queues, search indexes, file storage, analytics pipelines, authentication

services, monitoring tools, and external integrations. The challenge is not naming these components; the challenge is assigning responsibility clearly.

For example, a cache should not become the source of truth. A queue should not hide failed business logic. A notification service should not decide business approvals. A reporting database should not be used to execute transactional writes. Clear responsibility boundaries reduce future confusion.

1.6 Scalability, Reliability and Availability

The ability of the system to accommodate increases in users, traffic, data, regions, tenants, or process volume is known as scalability. Reliability is the ability of the system to carry out its intended function accurately and consistently over time under specified circumstances.

When a system is available, it can be used and accessed when needed. Although they are connected, these ideas are not the same.
If a system answers with inaccurate data, it could be accessible yet untrustworthy.

In a small setting, a system might be dependable, but under high load, it might not be scalable. If a single dependence failure knocks down a system, it may scale yet still have low availability. Each goal is defined independently in good design, and the system is tested against these goals.

1.7 Security and Data Architecture

Security includes

- authentication
- authorization
- encryption
- audit logging
- input validation
- secret management
- secure development practices
- monitoring for misuse.

A practical rule is least privilege: users, services, and jobs should have only the access they need. Another safe rule is deny by default: access should be explicitly allowed, not accidentally open.

Data architecture asks who owns each data object, where it is stored, how it is kept accurate, how long it is retained, who can access it, and how it moves between systems. This matters because most serious production failures involve either data inconsistency, data leakage, or data that cannot be trusted.

1.8 Integration Design

Seldom do modern systems operate independently.

- Payment gateways
- ERP systems
- HRMS platforms
- CRM systems
- mapping services
- Identity suppliers
- Analytics tools
- Notification providers
- Third-party APIs

are all connected to them.

- Latency
- failure modes
- security issues
- vendor dependence

are all increased with each integration.
Although it links the user experience to provider availability, a real-time payment integration provides quick feedback. Although it could be slower, a batch ERP integration is more manageable.

Clear contracts, retries, idempotency, timeouts, audit logs, and, when feasible, gradual degradation are all used in good integration design.

1.9 Cloud-Native Design

Cloud-native design is not simply “running on the cloud.” It means designing applications to take advantage of dynamic, automated environments.

Common techniques include

- Containers
- Orchestration
- managed services
- immutable infrastructure
- declarative APIs

- microservices where justified
- strong observability.

The goal is scalable, resilient, manageable software, not cloud complexity for its own sake.

Cloud services can reduce operational burden, but they also introduce

- cost management
- configuration risk
- service limits
- vendor lock-in
- new security responsibilities.

Cloud-native design still needs architecture discipline.

1.10 Observability

The capacity to comprehend a system's actions by looking at its outputs is known as observability. In actuality, this entails gathering and comparing company audit records, logs, metrics, traces, and events. Logs reveal the events.

Metrics display thresholds and trends. Request pathways between services are displayed by traces. Business decisions and actions are displayed in audit records.

Operating a system that lacks observability is challenging. The team may be aware that people are complaining, but they may not know which service is slow, which queue is backed up, which supplier is malfunctioning, or which deployment caused the issue.

1.11 Trade-Off Analysis

Common patterns include

- layered architecture
- modular monoliths
- microservices
- event-driven design
- CQRS
- Caching
- Queues
- read replicas
- API gateways.

These patterns are useful, but each has a cost. Microservices can improve independent deployment but add network complexity and distributed debugging. Event-driven design can decouple work but makes ordering, retries, and traceability harder. Caching can reduce load but introduces invalidation and stale-data risk.

The best architects do not ask, “Is this pattern popular?” They ask, “What problem does this pattern solve, and is that problem important enough to justify the cost?”

Example: Event Notification Platform

A mobile event notification platform needs

- location data
- event search
- recommendation ranking
- notification delivery
- user preferences
- organizer tools
- fraud prevention.

The system must balance freshness with battery life, personalization with privacy, and notification speed with spam prevention. That one example already touches requirements, data, security, scaling, and operations.

1.12 Common Mistakes

- Ignoring non-functional requirements until production.
- Overengineering before real scale requires it.
- Choosing tools because they are fashionable instead of justified.
- Failing to define source-of-truth ownership for data.
- Building without monitoring, alerting, and recovery plans.
- Treating security as a final checklist instead of an architectural concern.

1.13 Conclusion

System design is the discipline of making clear, practical trade-offs. A good design does not remove all complexity. It puts complexity in the right place, makes the system understandable, and gives teams a path to evolve safely.

Chapter 2 : Requirements Gathering

This chapter covers the concepts, patterns, trade-offs, architectural decisions and implementation concerns related to requirements gathering.

2.1 Introduction to Requirements Gathering

Serious design starts with requirements gathering. The team must comprehend the goals of the company, who will use the system, what it must perform, and how well it must function in production before selecting databases, services, APIs, or cloud technologies.

Weak requirements result in unnoticed rework. It's possible for a team to create the ideal technology for the incorrect issue. Alternatively, it can create the ideal feature but lack the functionality, security, or performance required for practical application. Strict constraints provide a foundation for architecture and minimize speculation.

2.2 Types of Requirements

Functional requirements describe system behavior:

- submit a form
- approve a request
- search an event
- process a payment
- generate a report
- or send a notification.

Non-functional requirements define quality:

- response time
- throughput
- availability
- reliability
- security
- maintainability
- usability
- compliance
- operational support.

Business requirements connect the project to outcomes such as

- reducing approval time
- improving conversion
- reducing operational cost
- or increasing auditability.

Operational requirements define deployment, support, backups, monitoring, incident response, and disaster recovery. Integration requirements define how the system interacts with external or internal systems.

Requirement Type	Question It Answers	Example
Business	Why are we building this?	Reduce reimbursement processing time from five days to one day.
Functional	What must the system do?	Allow employees to submit claims with attachments.
Non-functional	How well must it work?	Handle 10,000 monthly claims with 99.9% service availability.
Security	What must be protected?	Only finance reviewers can approve claims above the threshold.
Operational	How will it run?	Alert support when approval queues exceed SLA.
Integration	What systems must connect?	Sync approved claims to ERP finance module.

2.3 Stakeholder Analysis

Stakeholders include more than end users. They include executives, managers, operators, support teams, security teams, compliance teams, auditors, customers, vendors, and external systems.

- Each stakeholder has a different concern.
- Executives care about outcomes and cost.
- Users care about usability.
- Operations teams care about supportability.
- Security teams care about risk.
- Auditors care about traceability.

Missing a stakeholder often creates late surprises. For example, a reimbursement workflow may satisfy employees and managers but fail finance audit because it does not record policy evidence and reviewer comments. That is not a coding failure; it is a requirements failure.

2.4 Requirement Elicitation Techniques

How to gather these requirements?

- Interviews=> Explicit expectations are revealed during interviews.
- Workshops=> Conflicts are brought up in workshops

- Observation=> Hidden workarounds are discovered through observation
- document analysis=> helps teams understand existing policies, reports, forms, manuals, contracts, logs, and system records so they do not depend only on verbal explanations.
- process mapping=> helps stakeholders see how work actually moves from one person, system, or approval step to another, making bottlenecks and unclear ownership easier to identify.
- Surveys=> help collect input from a larger group of users or stakeholders, especially when interviews alone cannot cover enough people or perspectives
- prototypes= Prototypes enable stakeholders to respond to something tangible.
- and system analysis=> which helps teams study the current software, integrations, data flows, limitations, and failure points before deciding what the new or improved system should do.

The above are all used to find requirements. A single method is **insufficient**

Combining methods is a useful strategy. Business objectives should come first, followed by

- key stakeholder interviews
- process observation
- process mapping
- scenario creation
- edge case validation
- documentation of acceptance criteria.

2.5 Business Process Understanding

Understand a process before automating it. Bad workflows are inadvertently automated by many teams. They speed up a convoluted, slow process, but they don't improve it. Bottlenecks, manual handoffs, needless approvals, redundant data entry, ambiguous ownership, and missing audit points can all be found with the aid of process analysis.

For instance, creating a quicker form won't address the true issue if a purchase request takes ten days due to an unclear approval hierarchy. Approval guidelines, escalation routes, thresholds, and reviewer accountability must all be specified by the system.

2.6 Requirement Documentation

Requirements should not live only in meetings and chat messages. Useful documentation may include

- business requirement documents

- user stories
- use cases
- workflow diagrams
- data definitions
- acceptance criteria
- API assumptions
- integration contracts
- non-functional targets.

Good documentation is not bureaucracy; it is shared memory.

Each requirement should be clear, testable, and traceable. If a requirement cannot be tested or reviewed, it is probably too vague. “The system should be fast” is weak. “The search API should return results within 500ms for 95 percent of requests under expected load” is usable.

2.7 Functional Requirement Patterns

A lot of functional requirements fit into common patterns: stuff like

- Workflows
- Approvals
- event triggers
- reports
- notifications

or just classic input-processing-output.

Once you spot these patterns, everything gets faster. You can standardize across projects, reuse blocks, and set up your architecture without starting from scratch each time. These patterns also make testing and maintenance easier since people know what to expect.

If you catch the patterns early, you keep your design consistent and easier to manage. Most enterprise systems are just clever arrangements of a handful of these basic patterns anyway.

2.8 Non-Functional Requirements

Non-functional requirements shape architecture deeply. A system can have all requested features and still fail if it is slow, insecure, unreliable, hard to operate, or impossible to change. Strong non-functional requirements are measurable.

- Performance: response time, throughput, and latency targets.
- Scalability: expected growth in users, requests, data, regions, or tenants.
- Availability: percentage of time the service should be usable.

- Reliability: correctness and consistency of behavior over time.
- Security: identity, access control, encryption, audit, privacy, and threat controls.
- Maintainability: ease of change, testing, documentation, and modularity.
- Observability: logs, metrics, traces, alerts, dashboards, and audit records.
- Operability: deployment, rollback, backup, support, and disaster recovery.

2.9 Architectural Impact of Requirements

Architecture is directly impacted by requirements. Redundancy and failover may be necessary for high availability. Caching, indexing, data locality, or asynchronous processing may be necessary for low latency.

Immutable logs and reason codes may be necessary for strict audit needs. Data masking, encryption, permission checks, and centralized identification may all be necessary for strict security.
Every significant design decision should be based on a need or a purposeful compromise. A component may be needless complexity if a team is unable to justify its existence.

2.10 Trade-Off Analysis

Requirements often conflict. Better security can reduce convenience. Lower latency can increase cost. Faster delivery can reduce flexibility. High availability can increase operational complexity. Trade-off analysis makes these tensions explicit so stakeholders can choose with awareness.

The architect's role is not to promise everything. It is to make consequences visible. "We can support near-real-time notifications, but it will increase provider cost and operational monitoring. Or we can use periodic notifications and reduce cost, with less immediacy." That is a useful design conversation.

2.11 Conclusion

At the end of the day, getting requirements right lays the foundation for everything—architecture, development, testing, deployment, and smooth operations. Strong requirements make projects more predictable and less risky. They help align teams and guide decisions. Gathering requirements is both an art and a science, and investing in it delivers better results. The most successful systems start with clear, validated requirements, not guesses.

Chapter 3 : Functional vs Non-Functional Requirements

This chapter covers the concepts, patterns, trade-offs, architectural decisions and implementation concerns related to functional vs non-functional requirements.

It includes design considerations, scalability implications, operational concerns, security considerations and areas for future expansion.

3.1 Functional Requirements

Functional requirements define what the system must do. They describe

- Features
- Workflows
- Calculations
- Reports
- Notifications
- Validations
- user actions.

They are visible to stakeholders because they map directly to work people need to perform.

Examples include

- user registration
- password reset
- event search
- loan processing
- payroll execution
- inventory update
- report generation
- invoice approval
- ticket creation.

A strong functional requirement includes input, processing rule, output, exception behavior, and acceptance criteria.

Example: Loan Approval

A loan approval system may have functional requirements such as

- accepting an application
- verifying identity
- collecting income documents
- checking credit history
- applying scoring rules
- routing approval
- generating a decision
- logging the decision.

These are features and workflow steps. They define what the system does.

But finance systems also need non-functional requirements:

- auditability
- security
- explainability
- performance
- data retention
- regulatory compliance
- human review.

These requirements may be less visible to the user, but they are essential for safe operation.

3.2 Non-Functional Requirements

Non-functional requirements define the quality and constraints of the system. They answer questions such as:

- What is the optimal speed?
- How many users can it accommodate?
- How easily accessible must it be?
- How secure is the data?
- How easy is it to change?
- How will the group assess its well-being?

Specific examples are better than vague statements. "The system must be secure" is not enough. "A stronger statement might be "All sensitive personal data must be encrypted both in transit and at rest, access must be role-based, and all approval actions must be audit logged with user ID, timestamp, and reason code."

3.3 Requirement Patterns

Reusable structures known as requirement patterns can be found in a variety of systems. Common themes include

- notification systems
- event triggers
- reporting
- dashboards
- data synchronization
- exception handling
- audit trails,
- approval workflows.

You can design more quickly and reliably if you are aware of them.

But a pattern is not a solution that can be copied and pasted. A leave approval process differs from a high-value payment approval process. Although the structure may be identical, there are differences in risk, evidence, reviewer function, and audit criteria.

3.4 Architectural Impact

Functional requirements influence

- Services
- APIs
- data models
- user interfaces
- workflows.

Non-functional requirements influence

- infrastructure
- security controls
- monitoring
- scaling strategy
- data partitioning
- deployment model
- disaster recovery
- operational support.

For example, a functional requirement may say users can upload receipts.

A non-functional requirement may say uploaded receipts

- must be scanned for malware
- stored securely
- encrypted
- retained for seven years, and accessible only to authorized finance roles.

That changes storage, security, background processing, and audit design.

3.5 Scalability Considerations

Scalability requirements should include numbers. Ask what grows:

- concurrent users
- transactions per second
- data size, file volume
- tenants
- regions
- reporting complexity.

A vague requirement like “must scale well” does not help architecture. A measurable statement lets teams test and plan.

Designing too small causes outages. Designing too large too early wastes money and adds complexity. Good scalability planning uses business forecasts and clear growth assumptions.

3.6 Security Considerations

Early identification of security requirements is important.

- Authentication
- Authorization
- Encryption
- Audit
- Privacy
- secure logging
- input validation
- data masking
- secret management

- compliance are some of them.

Architecture, programming, testing, operations, and governance are all impacted by security.
Sensitive actions should always verify authorization, as a general rule. It is not authorized for a user interface to conceal a button. Access rules must be enforced by the backend.

3.7 Operational Concerns

Operational requirements define how the system will be run after launch. They include

- Deployment
- Monitoring
- Alerts
- Backups
- incident response
- support roles
- disaster recovery
- release management
- rollback.

A system is not production-ready simply because the feature works locally.

3.8 Design Decisions

Design decisions convert requirements into technical choices:

- database type
- API style
- cache strategy
- queueing approach
- deployment architecture
- authentication method
- monitoring stack.

These decisions should be documented with the reason, alternatives considered, and expected consequences.

3.9 AI System Requirements

AI systems require additional care. They need

- evaluation cases
- prompt and instruction governance
- human override rules
- traceability
- data quality checks
- model monitoring
- safe failure behavior.

If AI is used for decision support, the system must explain where the decision came from and when a human must review it.

3.10 Loan Approval Engine Example

Or check out a platform for loan approval. Business-side requirements include

- evaluating candidates
- assigning scores
- obtaining approvals
- producing reports.

However, you also need to justify your choices, pass audits, safeguard the data, work quickly, and adhere to rules. These encourage you to use audit logs, graph databases, and rule engines. These "non-functional" needs are just as important in finance as the essential elements.

3.11 Requirements Review Framework

A solid review process is a good compliance practice. Each review should check for clear business goals, a full and understandable spec, testability, and a clear path from requirements to results. Reviews work best when both technical folks and business stakeholders join in.

They're where you catch conflicts, clear up confusion, and see what's missing. Over time, this routine makes your governance and accountability stronger—plus, it mitigates risk and sets projects up for better delivery.

3.12 Best Practices

So work together to pin down requirements, check your assumptions early, write down decisions, and link everything back through the project's life.

Try out different ways of gathering feedback to really understand what's needed. Requirements should be clear, measurable, and testable—and you need both kinds right from the start.

Review them often as the project evolves. Strong governance and pulling in the right stakeholders really does pay off.

3.13 Common Mistakes

- Writing only functional requirements and ignoring quality attributes.
- Using vague phrases such as fast, secure, scalable, and user-friendly without measurable criteria.
- Missing acceptance criteria.
- Ignoring operational requirements until production.
- Letting scope creep enter without impact analysis.
- Treating non-functional requirements as optional extras.

3.14 Conclusion

Functional requirements deliver business features. Non-functional requirements keep those features safe, usable, scalable, reliable, and maintainable. Serious system design needs both from the beginning.

Chapter 4 : High-Level Architecture

This chapter covers the concepts, patterns, trade-offs, architectural decisions and implementation concerns related to high-level architecture.

4.1 Introduction to High-Level Architecture

High-level architecture is the broad blueprint of a system. It shows the major components, their responsibilities, how they communicate, where data lives, what systems are external, and how the system supports business goals.

It is not detailed code design, but it should be concrete enough for business, engineering, security, data, and operations teams to discuss trade-offs.

4.2 Purpose of High-Level Architecture

The purpose is alignment. Business teams understand what capability is being built. Developers understand the main components.

Operations teams understand deployment and support concerns.

Security teams can see trust boundaries. Data teams can see ownership and movement of information. A good high-level architecture reduces surprises before implementation begins.

4.3 Architecture Building Blocks

Common building blocks include

- user interfaces
- APIs
- business services
- databases
- caches, queues
- search indexes
- file stores
- analytics systems
- identity providers
- security controls
- monitoring tools
- external integrations.

Each block should have a clear responsibility.

Note that **Ambiguous responsibility is a design smell.**

For example,

- an order service should not also own payment-provider reconciliation unless that is an intentional design decision.
- A reporting database should not become the write path for business transactions.
- A cache should not be treated as permanent storage.

4.4 Architectural Views

Diverse stakeholders require various perspectives.

Workflows and capabilities are displayed in a business view. Components and duties are displayed in a logical view. Data ownership and flow are displayed in a data view.

External systems and communication are displayed in an integration view. Runtime infrastructure is displayed in a deployment view.

Access control and trust limits are displayed in a security view.
There isn't a diagram that works for everyone. The appropriate view is used for the appropriate audience in a good architectural package.

4.5 Common Architectural Patterns

Data access, business logic, and display are all kept apart via layered design. Modular monoliths enforce internal boundaries while maintaining a single deployable unit.

Microservices divide tasks into services that can be independently deployed. Producers and consumers are separated through the usage of events in event-driven architecture. When reads and writes have various requirements, CQRS distinguishes between command models and query models.

Every pattern generates new issues while resolving existing ones. Monoliths are not always superior to microservices. Systems that are event-driven are not inherently more scalable.

The team's maturity, business needs, operational capacity, and frequency of change must all be taken into consideration while choosing a pattern.

4.6 System Boundaries

A system boundary defines what the team owns and what it depends on.

Boundaries affect

- Security
- Integration
- source-of-truth decisions
- testing

- deployment
- support.

Weak boundaries create confusion: one team changes a field and another system breaks because nobody owned the contract.

Boundaries should be explicit around

- data
- APIs
- external providers
- user roles
- administrative operations
- audit responsibility.

4.7 Service Decomposition

Choosing how to divide duties is known as service decomposition. Good decomposition comes after ownership and business capabilities. Inadequate decomposition either produces too many tiny services before the team can run them or blindly follows technological layers.

Identifying business objects and workflows, such as

- Users
- Orders
- Payments
- Inventories
- Alerts
- Reports
- Approvals
- Claims
- Tickets

is a useful place to start. Next, find out which objects have different ownership or risk, which change concurrently, and which require separate scaling.

4.8 Data Architecture Considerations

Data ownership must be specified in high-level architecture. The client record is owned by which system? The payment status is owned by which system? Which system is in charge of the approval state? The audit history is owned by which system? Teams produce contradictory copies and lose faith in data when ownership is lacking.

Consistency, retention, privacy, reporting, lineage, and lifecycle are all taken into account in data architecture. Caches, search indexes, data lakes, analytics systems, and transactional systems all have distinct functions. They are not to be confused.

4.9 Integration Architecture

Very few systems live on their own anymore. Integration is about how you glue systems together — through APIs, messaging, events, or even old-school file exchanges.

This design should match how the business actually needs to operate. Bad integration choices create tech debt fast. Spell out who owns what, which protocols you'll use, and how you'll handle errors and failures. Well-designed integration is the heart of a flexible, future-ready system.

4.10 Scalability Considerations

Scalability has to be planned up front. Think about how the user base might grow, how much data you'll have, and what happens if things get way busier. This affects how you build your infrastructure, databases, and APIs.

Spot the Bottlenecks

Spot bottlenecks early and have a game plan. Getting this right means you avoid expensive changes down the road, and you can scale up (or out) when the business grows.

4.11 Reliability and Availability

Reliability and availability should be designed deliberately. Reliability focuses on correct behavior over time. Availability focuses on whether the system is usable when needed.

High availability may require

- Redundancy
- Failover
- health checks
- backups
- disaster recovery.

Reliability may require validation, retries, idempotency, reconciliation, and controlled error handling.

4.12 Security Architecture

Security architecture includes

- Identity
- access control
- encryption

- audit
- secret handling
- network boundaries
- secure configuration
- data classification
- threat modeling.

Security must be part of the diagram, not a separate afterthought.

4.13 Operational Architecture

Operational architecture covers

- Deployment
- Monitoring
- logging, tracing
- alerting
- incident response
- backup
- restore
- scaling
- rollback.

A system that cannot be operated safely is not a complete design.

4.14 Cloud-Native Architecture

Cloud-native architecture can use

- Containers
- Orchestration
- managed services
- declarative infrastructure
- automated deployment
- resilient service patterns.

But the goal is not to use every cloud feature. The goal is to build applications that can be deployed, scaled, observed, and recovered effectively in dynamic environments.

4.15 Trade-Off Analysis

Every architectural choice comes with trade-offs. Push hard for scalability, and things might get complicated. beef up security, and you risk making systems harder to use. Try to save money, and you might box yourself in.

Make decisions in light of your actual business needs — document the alternatives you considered and why you made a call. This way, future teams can understand, and you don't end up arguing the same points over and over.

4.16 Architectural Decision Records

Architectural Decision Records capture important choices: what was decided, why, what alternatives were considered, and what consequences are expected. Architectural Decision Records (ADR) prevent teams from repeatedly debating the same decision and help future maintainers understand context.

4.17 Conclusion

High-level architecture gives the system a shared shape. It does not eliminate complexity, but it makes complexity visible, discussable, and governable. A strong High-Level Design connects business goals with technical structure and future evolution.

Chapter 5 : Low-Level Design

5.1 Introduction to Low-Level Design

Low-Level Design, or Low-Level Design, turns high-level architecture into implementable detail. It defines

- APIs
- Classes
- Methods
- Schemas
- validation rules
- internal flows
- error handling
- data access behavior.

It is where “we need a reimbursement service” becomes actual request formats, database tables, service methods, permissions, and business rules.

5.2 High-Level Design vs Low-Level Design

High-level design answers: What major components exist, and how do they work together? Low-level design answers: How exactly will each component behave internally? **High-Level Design is the city map. Low-Level Design is the building plan.**

Both must stay aligned. If High-Level Design changes from monolith to services, Low-Level Design must change. If Low-Level Design reveals that the proposed boundary creates too much duplication or coupling, High-Level Design may need revision.

5.3 Purpose of Low-Level Design

The purpose of Low-Level Design is to reduce ambiguity before coding.

Developers should understand the

- object model
- API contracts
- validation rules
- persistence model
- error behavior
- test expectations.

Testers should know what scenarios to verify. Reviewers should know where business rules are enforced.

5.4 Core Building Blocks

Common Low-Level Design building blocks include

- Classes
- Interfaces
- Data Transfer Objects (DTO)
- Repositories
- Services
- Controllers
- Validators
- Mappers
- domain objects
- database tables
- API endpoints
- background jobs
- configuration.

Each building block should have a clear role. A class that validates input, sends email, writes records, and calculates pricing is trying to do too much.

5.5 Object-Oriented Design Principles

OOP is very important and systems which are to be built must try to follow these principles as much as possible so as to make them robust, observable and at the same extensible and flexible.

Encapsulation keeps internal details hidden.

Abstraction exposes useful behavior without leaking unnecessary implementation.

Inheritance and polymorphism can help model shared behavior, but they should not be overused.

5.6 SOLID principles

The **SOLID principles** are five essential guidelines for object-oriented programming created by Robert C. Martin to make software maintainable, scalable, and easy to understand.

The five fundamental principles are summed up as follows:

- S (Single Responsibility): A class ought to have just one task and one cause for modification. Without overlapping or disjointed logic, the code stays extremely cohesive.
- (Open/Closed): Software entities must be closed to alteration yet open to extension. New features can be added without disrupting or changing the current code.
- L (Liskov Substitution): Subtypes must be fully interchangeable with their base types. A kid class ought to act precisely what the parent class anticipates.
- I (Interface Segregation): No client should be compelled to rely on techniques that it does not employ. Large, all-purpose interfaces should be divided into more focused, smaller ones.
- D (Dependency Inversion): High-level modules must rely on abstractions rather than specifics. This separates your main reasoning from particular tools or low-level elements.

SOLID principles are useful when applied thoughtfully: single responsibility, open-closed design, substitutable abstractions, focused interfaces, and dependency inversion.

The aim is not to **worship principles**. The aim is maintainable code that can be tested and changed without causing unexpected damage. They should act more as guidance to good practices.

5.7 Onion Architecture

It is a system design pattern that keeps the business/domain logic at the center of the application.

The fundamental domain elements, rules, and business logic are found in the innermost layer.

The application layer, which organizes use cases and workflows, is positioned around the domain layer.

Databases, files, email, APIs, and external services are examples of infrastructure issues that remain in the outer layers.

Dependencies point inside rather than outward, which is the fundamental norm.

Thus, frameworks, databases, and user interface code shouldn't be a part of the domain logic.

For instance, a LeaveRequest rule shouldn't be aware of whether data is kept in an API, MongoDB, or PostgreSQL.

Because core business logic may be tested without the need for external systems, Onion Architecture enhances testability

It also improves **maintainability**, because changing the database or UI does not force major changes in business rules.

In system design, Onion Architecture is useful when you want a clean separation between **business logic, application flow, and technical infrastructure**.

5.8 Domain Modeling

Domain modeling represents the business in software. In a reimbursement system, domain concepts may include Employee, Claim, ExpenseLine, Receipt, Approval, PolicyRule, ExceptionRequest, and AuditEvent. A good model makes business rules visible instead of scattering them across random controllers and UI code.

Entities have identity and lifecycle. Value objects represent descriptive values such as Money, DateRange, Address, or ApprovalLimit. Domain services hold logic that does not naturally belong to one entity. This structure helps reduce duplication and confusion.

5.9 Class Design

Good class design keeps responsibilities focused. A ClaimValidator validates claims. A ClaimRepository persists claims. A ClaimApprovalService handles approval workflow logic. A NotificationService sends notifications. When classes become too large, testing becomes harder and changes become risky.

5.10 API Design Considerations

Application Programming Interfaces (APIs) serve as binding contracts between service providers and consumers. Because these guidelines establish reliable communication expectations, proper **endpoint names** and **request payloads** are essential to maintain API stability and prevent unexpected integration failures.

- Endpoint names
- request payloads
- response payloads
- status codes
- error codes
- validation rules
- authorization checks
- authentication requirements
- idempotency behavior
- versioning strategy

should all be specified by Low-Level Design. APIs ought to be reliable and consistent.

For instance, if the endpoint name indicates "create draft," then establishing a reimbursement claim should not be silently sent. Since submission may result in approval, locking, notification, and audit events, draft generation and final submission should frequently be done separately.

5.11 Database Schema Design

Database schema design translates business objects into reliable storage. It defines

- Tables
- Columns
- Types
- Constraints
- Indexes
- Relationships
- migration strategy.

A good schema protects data quality. Constraints are not optional decoration; they prevent invalid states from entering the system.

Normalization reduces duplication and update anomalies. Denormalization may improve read performance but introduces synchronization risk. The right balance depends on workload, reporting needs, and consistency requirements.

5.12 Design Patterns

Reusable solutions to recurrent design issues are known as design patterns. Complex construction rules can be created with the aid of a factory. Without big conditional blocks, a strategy can change its behavior. Persistence details can be concealed by the repository. Complex item creation can be made simpler with Builder.

Changes in status can be published by the observer. Code may be tested and configured more easily with Dependency Injection.
Instead of using patterns just because they sound professional, they should be employed because they address an actual issue. Overuse of patterns leads to confusion and formality.

5.13 Validation and Business Rules

Validation ensures inputs and state changes are allowed. Basic validation checks format, required fields, ranges, and types. Business validation checks rules such as “a claim above the receipt threshold must include a receipt” or “a manager cannot approve their own high-value purchase request.”

Business rules should be centralized enough to avoid duplication, but not so centralized that every change becomes difficult. They should be testable and traceable to requirements.

5.14 Error Handling Strategies

- Validation errors
- authorization errors

- not-found errors
- conflict errors
- dependency timeouts
- unexpected failures

should all be distinguished in error handling. User-facing communications should be beneficial without disclosing private company information.

Enough data should be kept in internal logs for troubleshooting purposes.
Retries ought to be utilized with caution. Generally speaking, it is safer to retry a read action than a write operation.

To prevent producing duplicate records, write operations require idempotency keys or duplicate prevention.

5.15 Scalability Considerations

Low-level design affects scalability through

- query design
- indexes
- pagination
- bulk operations
- caching hooks
- asynchronous jobs
- connection usage
- object loading patterns.

A system can have a good high-level architecture and still perform poorly because Low-Level Design creates expensive queries or loads too much data at once.

5.16 Security Considerations

Security at the Low-Level Design level includes

- input validation
- output encoding
- authorization checks
- secure password handling
- token validation
- secrets management

- safe logging
- encryption decisions
- prevention of injection attacks.

The design should specify where authorization is enforced, not merely assume the UI will hide restricted actions.

5.17 Performance Optimization

Performance optimization should be guided by expected bottlenecks and measurement.

Low-Level Design can improve performance by avoiding

- unnecessary database calls
- using indexes
- reducing payload size
- caching safe data
- batching operations
- avoiding expensive synchronous work in user-facing requests.

Premature optimization can make code harder to maintain, so optimize where evidence or requirements justify it.

5.18 Testing Considerations

A good low-level design is testable. Unit tests verify small pieces of logic. Integration tests verify component behavior with databases or external systems. Contract tests verify API compatibility. Performance tests verify non-functional requirements. Security tests verify access and input handling. Testability is not accidental; it is designed.

5.19 Operational Concerns

Low-Level Design should include

- logging points
- metrics
- trace IDs
- audit events
- feature flags
- configuration
- safe deployment behavior.

Operational design helps support teams diagnose problems without reading code at midnight.

5.20 Trade-Off Analysis

Low-level design also involves trade-offs. More abstraction can improve flexibility but reduce readability. More validation can improve safety but increase development effort. More normalization can improve data quality but make queries more complex. Good Low-Level Design documents the reason behind important choices.

5.21 Future Expansion Planning

Future-ready LLD uses clear interfaces, modular design, configuration for expected variation, migration-friendly schemas, and documented extension points. But future planning should not become overengineering. Design for likely change, not every imaginable change.

5.22 Conclusion

Low-level design turns architecture into software that developers can build and maintain. It protects the system from ambiguity. When LLD is clear, implementation becomes faster, testing becomes easier, and long-term maintenance becomes safer.