How to become a Staff AI Engineer

Chapter 2: Software Engineering Foundations for AI Engineering

Celso Sousa
LinkedIn: https://www.linkedin.com/in/celso-sousa/?locale=en-US
Portfolio: https://celsosousa.com.br/project-portfolio/
Ebook: https://celsosousa.com.br/ebook-staff-ai-engineer/

Chapter Objective

Modern AI systems are, first and foremost, software systems. Models, agents, retrieval pipelines, evaluation mechanisms, and observability components only generate value when they are integrated into an architecture that can evolve without becoming fragile, expensive, or impossible to maintain.

For a Staff AI Engineer, Software Engineering is not a complementary skill. It is the discipline that transforms probabilistic capabilities into reliable operational systems.

The primary objective of this chapter is to develop the ability to evaluate not only whether a system works today, but whether it will continue to work when:

- the number of users increases;
- new models are incorporated;
- business rules change;
- providers are replaced;
- new security requirements emerge;
- multiple teams begin modifying the same platform.

The concepts presented here should be interpreted as tools for controlling complexity.

- **Software design reduces the cost of change:** A good structure allows models, pipelines, or providers to be replaced without rebuilding the entire application.
- **Architecture defines responsibility boundaries:** Well-separated components reduce failure propagation and make evolution easier.
- **Patterns make recurring decisions explicit:** Instead of reinventing solutions, patterns provide known ways to organize behavior.
- **Code quality preserves future velocity:** Code that is difficult to understand turns small changes into expensive projects.
- **Versioning reduces operational risk:** Changes to code, models, prompts, and APIs must be traceable and reversible.

At the Staff level, the goal is not to apply every Software Engineering principle indiscriminately. It is to use enough abstraction to reduce risk and complexity without creating overengineering.

2.1. Software Design

Software design defines how responsibilities are distributed within a system. In AI Engineering, this is especially important because an application may combine deterministic logic, probabilistic models, databases, external APIs, tools, and security mechanisms.

When these responsibilities are mixed together, any change can affect multiple parts of the system. When they are properly separated, the architecture can evolve with significantly less risk.

2.1.1. Separation of Concerns

Separation of Concerns means dividing a system according to distinct responsibilities. In an AI product, response generation, authentication, retrieval, authorization, observability, and model access represent different concerns. Combining them within a single component creates strong dependencies among parts that evolve for different reasons.

The primary benefits appear in maintainability and security.

- **Each component has a clear responsibility:** Changes to retrieval logic do not need to affect authentication or business rules.
- **Problems become easier to diagnose:** If quality degrades, retrieval, reranking, generation, or post-processing can be investigated independently.
- **Tests become more specific:** Each layer can be validated in isolation before end-to-end evaluation.
- **Security controls become more explicit:** Authorization can be verified in components that are independent of the model.
- **Teams can work with less interference:** Different teams can evolve parts of the platform through well-defined contracts.

Consider an enterprise agent capable of accessing financial information and generating reports. A poor implementation might allow the same component that interprets the request to also determine permissions, access data, and execute actions. This creates a serious problem: probabilistic logic begins controlling functions that should remain deterministic.

A safer architecture separates:

- **Intent interpretation:** May use models.
- **Authorization:** Must follow deterministic policies.
- **Data access:** Must respect explicit permissions.
- **Action execution:** Must validate limits and business rules.
- **Auditing:** Must record what happened independently of model behavior.

The practical rule is simple: components that have different reasons to change should generally be separated. However, excessive separation also has a cost. An extremely fragmented system can introduce too many

interfaces, network calls, and operational complexity. The goal is to find a decomposition that improves clarity without turning every small function into an independent service.

2.1.2. Modularity

Modularity represents the ability to divide a system into units that can evolve relatively independently. In AI Engineering, modularity is particularly valuable because the ecosystem changes rapidly. Models, providers, embedding mechanisms, and agent frameworks may be replaced within a matter of months.

A modular architecture reduces the cost of these changes.

- **Models can be replaced with less impact:** The application depends on a stable interface rather than a specific provider.
- **Retrieval can evolve independently from generation:** The search strategy can be changed without rewriting the application.
- **Components can be tested independently:** This improves diagnosis and development velocity.
- **Capabilities can be reused:** An evaluation or authentication module can serve multiple applications.
- **Vendor lock-in is reduced:** External dependencies remain encapsulated within specific modules.

Imagine a company that directly integrates an LLM provider in dozens of places throughout its codebase. If the provider changes its API or the company decides to migrate to another model, many parts of the system must be modified.

A modular architecture creates an abstraction layer for model access. The application understands the capability it requires, while provider-specific details remain isolated. This creates an important strategic benefit: optionality.

The organization can experiment with new providers or architectures without incurring disproportionate migration costs. The trade-off is that abstractions also need to be maintained. Creating a generic interface before understanding the actual requirements can result in a weak or unnecessarily complex abstraction.

Modularity should be driven by likely points of change, not by an attempt to predict every possible future requirement.

2.1.3. Encapsulation

Encapsulation means hiding internal implementation details and exposing only what other components actually need to use. This principle reduces the amount of knowledge required to consume a capability.

In an embedding system, for example, the application should not need to know the details of batching, retries, tokenization, caching, or providers in order to request a vector representation. The responsible component should encapsulate those decisions.

The benefits are significant.

- **Reduces coupling:** Consumers depend on the contract rather than the internal implementation.
- **Makes changes easier:** Strategies can be changed without modifying consumers of the component.
- **Improves security:** Credentials and sensitive implementation details can remain hidden.
- **Reduces cognitive complexity:** Engineers can use a capability without understanding its entire implementation.

- **Enables centralized policies:** Rate limiting, logging, and authentication can be enforced within the layer itself.

Encapsulation is also important when interacting with LLMs. The application should not depend directly on the exact way a particular model represents tool calls, structured outputs, or errors.

An intermediate layer can normalize these behaviors. This makes provider replacement easier and reduces the propagation of changes. However, hiding too much information can harm observability. An encapsulated component must still expose enough metrics and information for troubleshooting.

Good encapsulation hides operational complexity without hiding the signals required to understand behavior.

2.1.4. Cohesion

Cohesion measures how closely related the responsibilities within a component are. High cohesion means that a component has a clear purpose. Low cohesion occurs when a component performs activities that are not directly related.

Imagine a service responsible for:

- preparing prompts;
- authenticating users;
- querying a database;
- calculating billing;
- recording audit information;
- calling the LLM.

This component has low cohesion. A change to any one of these responsibilities can affect everything else.

High cohesion provides important advantages.

- **Components become easier to understand:** Their purpose can be explained simply.
- **Changes become localized:** Modifications to one responsibility affect less code.
- **Tests become clearer:** The expected behavior of the module is well defined.
- **Ownership improves:** A team can take responsibility for a specific capability.
- **Failures become more contained:** Problems tend to remain within known boundaries.

A good example is a service dedicated to model routing. Its responsibility may be to select the appropriate model based on quality, cost, and latency requirements. If that service also begins managing billing, user authorization, and document ingestion, its cohesion decreases.

A Staff Engineer should continuously identify components that have accumulated responsibilities over time. This accumulation is usually a sign of future technical debt.

2.1.5. Coupling

Coupling describes the degree of dependency between components. Some coupling is inevitable. The problem emerges when a change in one component requires changes across many others. The general objective is high cohesion and low coupling.

There are several forms of problematic coupling.

- **Implementation coupling:** One component depends on another component's internal details.
- **Temporal coupling:** Two services must be available simultaneously for the system to function.
- **Data coupling:** Many services depend directly on the same internal schema.
- **Provider coupling:** The entire application depends on provider-specific behavior from an external API.
- **Organizational coupling:** One team must constantly wait for another team to make simple changes.

In AI systems, direct provider dependency is a recurring example. If the entire codebase understands the specific formats of a particular LLM, migrating to another provider becomes expensive.

An adaptation layer reduces this coupling. However, reducing coupling also has a cost. Messaging systems, abstractions, and intermediate interfaces add components and complexity. The architecture must balance independence with simplicity.

A small system may initially tolerate greater coupling. As it scales in functionality, team count, or criticality, reducing these dependencies becomes increasingly valuable.

2.1.6. SOLID

SOLID is a set of object-oriented design principles that helps reduce rigidity and improve maintainability. Although it originated in object-oriented programming, the principles represent ideas that are useful across architectural styles.

Single Responsibility Principle

A component should have one clearly defined primary responsibility.

- **Reduces the impact of changes:** Changes to authentication should not require changes to retrieval.
- **Makes ownership easier:** It becomes clearer which team maintains a given capability.
- **Improves testability:** Each component has more limited and predictable behavior.

In AI Engineering, this principle is particularly useful for separating probabilistic logic from deterministic rules.

Open/Closed Principle

Components should support extension without requiring constant modification of the existing core.

- **New models can be added through existing interfaces:** This reduces regression risk.
- **New ranking strategies can be plugged in:** The primary pipeline remains stable.
- **New providers can be incorporated:** The application does not need to be rewritten.

This principle is important in AI environments because new technologies emerge rapidly.

Liskov Substitution Principle

Different implementations of an abstraction must preserve consumer expectations. If multiple models implement the same interface, all of them need to honor the essential contract.

This is particularly important in model gateways. A model should not silently return incompatible behavior simply because it uses a different provider.

Interface Segregation Principle

Consumers should not depend on capabilities they do not use. A massive model interface may force simple systems to understand capabilities related to audio, images, tools, and streaming even when they only need text.

Smaller interfaces reduce dependencies and make evolution easier.

Dependency Inversion Principle

High-level components should depend on abstractions rather than directly on concrete implementations. A business application should depend on a generation capability, not directly on a provider-specific SDK.

This provides important architectural benefits.

- **Makes testing easier:** Implementations can be replaced with test doubles or mocks.
- **Reduces vendor lock-in:** Providers remain behind interfaces.
- **Makes experimentation easier:** New models can be compared.
- **Improves resilience:** Fallbacks become easier to implement.

SOLID should not be treated as dogma. Applying every principle to an extreme can create excessive abstraction and make the system harder to understand. The goal is to reduce coupling at points where change and complexity justify the additional structure.

2.2. Software Architectures

Architecture defines the high-level structure of a system, the boundaries between components, and the way those components communicate.

There is no universally superior architecture. Different styles solve different problems. The decision must consider scale, complexity, number of teams, isolation requirements, fault tolerance, and the expected rate of change.

2.2.1. Layered Architecture

Layered Architecture organizes the system into layers, each with specific responsibilities.

A typical application may separate:

- presentation;
- application logic;
- domain;
- data access;
- infrastructure.

In AI systems, a similar decomposition can isolate the interface, orchestration, domain logic, AI services, and infrastructure.

The primary benefits are simplicity and clarity.

- **Responsibilities are organized:** Each layer has a known function.
- **Dependency flow becomes predictable:** Higher-level components use services from lower layers.
- **Onboarding is easier:** Engineers can locate functionality more easily.
- **Testing can be performed by layer:** This reduces diagnostic complexity.
- **It works well for many enterprise products:** Not every application requires a sophisticated distributed architecture.

However, layers can become excessively coupled. When every request must traverse many layers, simple changes may require modifications in multiple places.

Another risk is allowing business logic to become overly dependent on database, cloud, or AI provider details. Layered Architecture is a good choice for applications whose complexity is still moderate and where operational simplicity has high value.

2.2.2. Hexagonal Architecture

Hexagonal Architecture, also known as Ports and Adapters, seeks to protect the core logic of a system from external dependencies. The central idea is to separate the domain from specific technologies.

The core defines what must be accomplished. Interfaces represent required capabilities. Adapters connect those interfaces to concrete implementations. This is particularly useful in AI Engineering.

An application may conceptually depend on:

- a text generator;
- a retrieval mechanism;
- a repository;
- an external tool.

These capabilities can be implemented by different providers without modifying the domain.

The primary benefits include:

- **Reduced technological dependency:** Cloud providers, databases, or models can be replaced.
- **Greater testability:** External dependencies can be simulated.
- **Greater domain stability:** Business rules do not change along with SDKs.
- **Better control over vendor lock-in:** Implementation details remain confined to adapters.
- **Greater architectural clarity:** The system distinguishes capabilities from implementations.

For AI systems, this makes it possible to replace an LLM provider while keeping the rest of the application relatively stable. The primary disadvantage is the additional number of interfaces and abstractions.

In small systems, this may seem unnecessary. Hexagonal Architecture provides greater value when external dependencies change frequently or when the domain is important enough to justify isolation.

2.2.3. Clean Architecture

Clean Architecture shares many principles with Hexagonal Architecture, emphasizing independence between business rules and external details. The goal is to protect the most important parts of the system from changes in frameworks, databases, interfaces, and infrastructure. In AI Engineering, this helps prevent business decisions from being embedded directly in prompts or integration code.

Several principles are especially important.

- **Domain rules should remain independent of models:** Critical rules should not exist only inside LLM instructions.
- **Use cases define application behavior:** The system should have an explicit layer that orchestrates objectives.
- **Infrastructure should be replaceable:** Databases, providers, and frameworks are implementation details.
- **Dependencies should point toward the core:** Core components should not know about external details.

This approach improves governance. If a regulatory rule states that a particular action requires human approval, that rule should not depend on the model behaving as expected. It should exist explicitly within the architecture.

Clean Architecture also makes auditing easier because critical responsibilities become more visible.

Once again, the risk is overengineering. Not every application requires multiple conceptual layers. Architecture should remain proportional to the size and criticality of the problem.

2.2.4. Event-Driven Architecture

Event-Driven Architecture organizes systems around events that represent important facts. Instead of components directly calling one another in every flow, events can be published and consumed asynchronously. This style is very useful when multiple capabilities need to react to the same occurrence.

Consider a new document entering a knowledge platform. The event could independently trigger:

- parsing;
- classification;
- metadata extraction;
- embeddings;
- indexing;
- auditing;
- notification.

The benefits are significant.

- **Reduces coupling between producers and consumers:** The producer does not need to know about every downstream reaction.
- **Makes scalability easier:** Consumers can scale independently.
- **Supports asynchronous processing:** Long-running operations do not block the user.
- **Makes evolution easier:** New consumers can be added without modifying the producer.
- **Improves resilience:** Queues can absorb temporary load spikes.

However, Event-Driven Architecture introduces its own problems.

- **Consistency may no longer be immediate:** Different components may observe state at different times.
- **Debugging becomes more complex:** A flow may traverse multiple services and queues.
- **Duplicate events can occur:** Consumers need to be idempotent.
- **Ordering may matter:** Some processes require sequence preservation.
- **Distributed observability becomes essential:** Without tracing, incidents become difficult to investigate.

In AI systems, events are particularly useful for ingestion pipelines, feedback collection, asynchronous evaluation, and index updates. They are not necessarily the best choice for every synchronous user interaction.

2.2.5. Microservices

Microservices divide an application into independent services, typically with separate deployment and ownership. The primary advantage is not technical, but organizational: different teams can evolve components with greater independence. This architecture may make sense when clear boundaries exist.

Possible examples in an AI platform include:

- ingestion service;
- retrieval service;
- model gateway;
- evaluation service;
- billing service;
- identity service.

Microservices can provide important benefits.

- **Independent scalability:** Retrieval and model serving may have very different scaling requirements.
- **Independent deployments:** A team can evolve its service without releasing the entire platform.
- **Fault isolation:** A failure may remain confined to one part of the system.
- **Clear ownership:** Teams can take responsibility for specific services.
- **Technological heterogeneity:** Different workloads can use appropriate technologies.

However, microservices turn local problems into distributed systems problems. The organization must now deal with:

- network failures;
- retries;
- service discovery;
- distributed tracing;
- eventual consistency;
- API versioning;
- distributed observability;
- greater deployment complexity.

For this reason, microservices should not be adopted simply because they are associated with large-scale architectures. A company can create dozens of services before it has the scale or organizational structure required to justify them. The result is a platform that is difficult to operate.

An important principle for interviews is:

Microservices should be justified by requirements for independence, scale, or ownership, not by architectural preference.

2.2.6. Modular Monoliths

A Modular Monolith combines relatively simple deployment with well-defined internal boundaries. The system operates as a single application, but is internally organized into independent modules.

This approach is often underestimated. It can be particularly appropriate for AI products in early or intermediate stages.

The benefits include:

- **Lower operational complexity:** There is no need to manage dozens of services.
- **Simpler transactions and consistency:** Many interactions remain local.
- **Easier debugging:** The flow occurs within the same application.
- **Less complex deployments:** Infrastructure overhead is lower.
- **Boundaries can be preserved:** Good modular design allows services to be extracted later.

The primary risk is allowing internal boundaries to deteriorate. If modules begin directly accessing one another's internal structures, the system becomes a tightly coupled monolith.

For this reason, modularity must be treated as a real architectural constraint. For many products, a Modular Monolith is an excellent initial architecture.

A Staff Engineer can preserve operational simplicity while maintaining the option to extract services when real requirements emerge. This approach reflects an important principle: **do not pay in advance for scaling complexity that does not yet exist**.

2.3. Relevant Design Patterns

Design patterns represent recurring solutions to design problems. The goal is not to memorize names. Their value lies in recognizing structural problems and understanding which mechanisms can reduce coupling or increase flexibility.

In AI Engineering, some patterns become particularly useful because of the variety of models, providers, tools, and strategies that need to be replaced or combined.

2.3.1. Factory

Factory centralizes the creation of objects or components. It is useful when the way a capability is instantiated varies depending on configuration or context.

In AI Engineering, it can be used to instantiate:

- different models;
- different embedding providers;
- different retrievers;

- different agents.

The benefits include:

- **Centralizes creation logic:** Specific configurations are no longer scattered throughout the codebase.
- **Makes implementation replacement easier:** The application requests a capability without knowing its construction details.
- **Improves testability:** Implementations can be replaced during testing.
- **Reduces provider dependency:** SDK usage remains concentrated in specific locations.

The primary risk is creating overly generic factories. If object creation is simple and stable, adding another layer may simply increase complexity. Factories are most useful when multiple implementations exist or when substantial configuration is required.

2.3.2. Strategy

Strategy makes it possible to encapsulate different algorithms or policies behind a common interface. This pattern is extremely relevant to AI Engineering.

A system may have different strategies for:

- model selection;
- retrieval;
- reranking;
- caching;
- fallback;
- evaluation.

For example, a router may use a strategy to determine which model will handle each request. One strategy may prioritize quality, another may prioritize cost, and another may prioritize latency.

The benefits include:

- **Makes experimentation easier:** Strategies can be compared without rewriting the system.
- **Reduces scattered conditionals:** Logic remains concentrated in specific components.
- **Enables context-specific adaptation:** Different customers may use different policies.
- **Makes testing easier:** Each strategy can be validated independently.

From a business perspective, Strategy allows economic policies to be directly incorporated into the architecture. An organization may use expensive models only when the expected benefit justifies the cost.

The risk is creating dozens of strategies without governance, making system behavior difficult to understand. Strategies need clear criteria and observability.

2.3.3. Adapter

Adapter converts the interface of an external implementation into an interface expected by the system. It is one of the most important patterns for reducing vendor lock-in. A company may define an internal interface for text generation and create provider-specific adapters for different providers.

The benefits include:

- **Isolates provider differences:** Provider-specific formats do not contaminate the rest of the codebase.
- **Makes migration easier:** The organization can replace providers.
- **Enables fallback:** Alternative implementations can be invoked.
- **Reduces the impact of external changes:** Changes to an SDK remain confined to the adapter.
- **Makes integration testing easier:** Each provider can be validated independently.

Adapter is also useful when integrating legacy systems. A new agent platform can access an older system through a modern interface without requiring the legacy system to be immediately rewritten. This creates a strategy for incremental modernization.

2.3.4. Observer

Observer allows components to react to changes or events without strong direct dependencies. It is appropriate for systems in which multiple actions need to occur after a particular event.

In AI Engineering, it can be used to:

- record metrics;
- collect feedback;
- initiate evaluations;
- trigger auditing;
- update dashboards.

The primary benefit is separating core behavior from auxiliary reactions. A model call, for example, may produce an event observed by components responsible for cost, telemetry, and quality monitoring. This reduces the need to embed observability logic throughout every execution flow.

The risk is implicit behavior. When many observers react to an event, it can become difficult to understand everything that occurs.

In complex systems, tracing and documentation become important for maintaining visibility.

2.3.5. Repository

Repository creates an abstraction between application logic and data persistence. Instead of the domain understanding specific details of SQL, vector databases, or storage APIs, it uses an access interface.

The benefits include:

- **Reduces dependency on storage technology:** Databases can be replaced.
- **Makes testing easier:** Persistence can be simulated.
- **Centralizes queries and access rules:** This reduces duplication.
- **Improves governance:** Access controls can be applied consistently.

In RAG systems, it is important to avoid turning repositories into excessively generic abstractions.

Vector search, lexical search, and transactional operations have different semantics. The abstraction should preserve relevant differences instead of artificially hiding them.

2.3.6. Dependency Injection

Dependency Injection provides a component's dependencies externally instead of allowing the component to create all of its own dependencies. This pattern reduces coupling and improves testability.

Consider a service that requires:

- a model;
- a retriever;
- a logger;
- a policy engine.

Instead of directly instantiating each implementation, those dependencies can be provided during initialization.

This provides important advantages.

- **Makes replacement easier:** Models and providers can vary across environments.
- **Improves testing:** Real dependencies can be replaced.
- **Makes dependencies explicit:** It becomes clear what the component requires.
- **Supports configuration:** Different deployments can use different implementations.
- **Reduces coupling:** The component does not depend directly on concrete classes.

Dependency Injection is particularly useful in AI Engineering because many components need to be experimented with and replaced frequently. However, excessively sophisticated dependency injection frameworks can make the construction flow difficult to understand. Simplicity should remain a priority.

2.4. Code Quality

Code quality should not be treated as an aesthetic concern. It has a direct impact on delivery velocity, incident volume, onboarding capacity, and maintenance cost. In organizations with multiple AI teams, code quality also affects governance because systems that are difficult to understand are more difficult to audit.

2.4.1. Maintainability

Maintainability represents the ease with which a system can be fixed, expanded, or adapted. This attribute becomes critical in AI because components change rapidly.

A maintainable codebase has several characteristics.

- **Clear boundaries:** Responsibilities are separated.
- **Controlled dependencies:** Changes do not propagate indiscriminately.
- **Reliable tests:** Changes can be made with less fear of regression.
- **Sufficient documentation:** Important decisions can be understood.
- **Adequate observability:** Problems can be diagnosed in production.

Maintainability has a direct economic impact. A system that is difficult to change increases:

- development time;
- incident cost;
- regression risk;
- dependency on specific specialists.

The last point is particularly dangerous. If only one person understands a critical part of the system, there is substantial key-person risk. A Staff Engineer should treat maintainability as a factor in operational sustainability.

2.4.2. Readability

Readability represents how easily the intent of code can be understood. Code is read far more often than it is written. For this reason, clarity usually provides more value than overly clever solutions.

Readability improves when:

- **Names communicate intent:** Components make their roles clear.
- **Functions have limited purposes:** Smaller units are easier to reason about.
- **Complex flows are explicit:** Important behaviors are not hidden.
- **Abstractions have real meaning:** Interfaces represent domain concepts.
- **Documentation explains decisions:** Comments should explain reasons rather than repeat the code.

In AI Engineering, clarity is also important for security. An authorization pipeline that is difficult to understand increases the risk of implementation errors. Similarly, critical policies hidden inside prompts can be difficult to audit. Readability should exist both in the code and in the architecture.

2.4.3. Technical Debt

Technical debt represents decisions that reduce effort in the short term in exchange for greater future cost. Not all technical debt is bad. In some situations, consciously taking on debt can be rational in order to validate a business hypothesis. The problem arises when the debt is no longer known or controlled.

There are several forms.

- **Code debt:** Duplicated or difficult-to-maintain code.
- **Architecture debt:** Inadequate boundaries or tight coupling.
- **Data debt:** Inconsistent schemas or poor data quality.
- **Infrastructure debt:** Manual processes or fragile environments.
- **AI debt:** Unversioned prompts, datasets, or evaluations.
- **Security debt:** Temporarily absent controls.
- **Governance debt:** Missing documentation or ownership.

In AI, debt can accumulate rapidly during prototyping. A project may begin as a demonstration and gradually acquire production users without going through an appropriate engineering phase. This process creates a dangerous phenomenon: **prototype-to-production drift**.

A Staff Engineer must identify when a proof of concept has moved beyond the context for which it was built. Technical debt should be prioritized based on impact rather than engineer preference. Problems that increase security risk, prevent scaling, or repeatedly delay new products deserve higher priority.

2.4.4. Refactoring

Refactoring changes the internal structure of software without modifying its expected external behavior. The goal is to improve the system's future capacity for change.

Refactoring may become necessary when signals such as the following appear:

- **Increasing duplication:** The same logic appears in multiple locations.
- **Excessively large components:** Too many responsibilities are concentrated in one place.
- **Simple changes affect multiple areas:** Strong coupling exists.
- **Testing is difficult:** The design does not support isolation.
- **Onboarding has become slow:** The architecture is difficult to understand.

In AI systems, refactoring often becomes necessary when a prototype evolves into a platform. A direct integration with a single LLM may need to evolve into a model gateway. A specialized retrieval implementation may need to become a reusable capability.

The primary trade-off is between engineering investment and delivery continuity.

A complete refactoring effort may interrupt product development for long periods. Incremental strategies are usually better. A Staff Engineer should seek structural improvements that can be introduced progressively while preserving delivered value.

2.4.5. Static Analysis

Static analysis examines code without executing it, identifying problems related to quality, security, or consistency. It is especially useful because it can automatically detect classes of errors before production.

Its primary functions include:

- **Type checking:** Reduces errors related to incorrect contracts.
- **Linting:** Identifies problematic patterns and inconsistencies.
- **Security scanning:** Detects known vulnerabilities or unsafe practices.
- **Dependency analysis:** Identifies vulnerable libraries.
- **Complexity analysis:** Helps identify components that are difficult to maintain.

In AI Engineering, static analysis complements but does not replace behavioral evaluation. A pipeline may be perfectly type-safe and still produce poor responses. AI systems therefore need to combine traditional Software Engineering controls with AI-specific evaluation.

From a governance perspective, automating checks within the CI pipeline reduces dependence on manual inspection. This makes it possible to transform standards into executable controls.

2.4.6. Code Review

Code Review is a tool for quality, knowledge sharing, and risk control. Its objective should not be limited to identifying syntax errors.

A mature review evaluates multiple dimensions.

- **Correctness:** Does the implementation actually solve the problem?
- **Architecture:** Does the change respect existing boundaries?
- **Security:** Have new attack surfaces been introduced?
- **Observability:** Will problems be diagnosable?
- **Maintainability:** Will the solution remain understandable?
- **Testing:** Are critical cases covered?
- **Systemic impact:** Does the change affect other components or teams?

In AI systems, code review should also consider changes that traditionally existed outside the code itself. Prompts, policies, evaluation datasets, and model configurations can significantly alter system behavior. These artifacts also require appropriate review.

A Staff Engineer should not turn review into a mechanism for personal centralization. The goal is to improve the quality of the entire team: **good reviews explain the reasoning behind decisions and help other engineers develop stronger decision-making capabilities**.

2.5. Versioning

Versioning is fundamental to traceability and reversibility. In traditional systems, we generally think about code versions. In AI Engineering, however, behavior also depends on models, prompts, datasets, configurations, indexes, and providers. Versioning therefore needs to be interpreted more broadly.

2.5.1. Git

Git records the evolution of code and enables multiple engineers to work in a coordinated way. Its value at the Staff level lies less in specific commands and more in the discipline of change management that it enables.

A high-quality history helps answer:

- who changed a particular piece of logic;
- why the decision was made;
- which version introduced a regression;
- which state can be restored.

In AI systems, code alone may not be sufficient to reproduce behavior. It may also be necessary to associate a version with:

- the model;
- the prompt;
- the dataset;
- the retrieval configuration;
- the evaluation suite.

True reproducibility depends on the combination of these artifacts.

2.5.2. Branching Strategies

A branching strategy defines how teams organize parallel development. There are both simpler and more structured approaches. The most important principle is to reduce the time between creating and integrating changes.

Long-lived branches increase divergence and the risk of conflicts. Modern practices often favor frequent integration.

The benefits include:

- **Faster feedback:** Problems emerge earlier.
- **Lower conflict risk:** Changes remain smaller.
- **More frequent deployments:** New functionality can reach users quickly.
- **Greater alignment:** The main branch remains close to the actual state of development.

In regulated environments, it may be necessary to separate releases or establish more rigorous approval processes. The strategy must reflect both risk and organizational context.

A Staff Engineer should avoid adopting complex processes when automated pipelines and feature flags already provide sufficient isolation.

2.5.3. Pull Requests

Pull Requests provide a mechanism for reviewing and integrating changes. In mature teams, PRs serve as units of technical communication.

A good change should allow the reviewer to understand:

- what problem is being solved;
- which alternatives were considered;
- which components were affected;
- how the change was tested;
- which risks remain.

Excessively large PRs are difficult to review. This increases the probability that errors will go unnoticed. Small, incremental changes generally provide better quality and velocity.

In AI Engineering, prompt or configuration changes may appear small in terms of lines changed while producing substantial behavioral differences. The size of the impact should not be measured solely by the size of the diff. Changes to AI behavior need to be accompanied by evaluation evidence.

2.5.4. Semantic Versioning

Semantic Versioning communicates the type of change that occurred in a component. The fundamental idea is to distinguish compatible changes from changes that break existing contracts.

This becomes important when platforms are consumed by multiple teams. If a corporate API or SDK silently changes behavior, dozens of applications may fail. Explicit versioning makes it possible to manage evolution.

In AI platforms, compatibility needs to account not only for API structure but also for behavior.

A new model may preserve exactly the same interface while still significantly changing:

- quality;
- latency;
- response format;
- tool calling;
- cost.

AI systems may therefore need to complement Semantic Versioning with behavioral evaluation. Syntactic compatibility does not guarantee semantic compatibility.

2.5.5. Release Management

Release Management governs how changes reach production. The objective is to balance velocity with risk control.

A mature process may include:

- **Automated checks:** Traditional tests and AI evaluations are executed before release.
- **Approval gates:** High-risk changes may require additional validation.
- **Canary rollout:** A small portion of traffic receives the new version.
- **Observability:** Metrics are compared against a baseline.
- **Rollback:** The previous version can be restored quickly.
- **Release notes:** Important changes are documented.

In AI systems, Release Management needs to include artifacts beyond code. A change may occur without any traditional deployment.

For example:

- the provider updates the model;
- a prompt is modified;
- new documents are added to the knowledge base;
- a vector index is rebuilt;
- an agent policy changes.

Any of these changes can alter system behavior. AI Engineering maturity therefore requires controlling behavioral changes, not merely code changes. This is particularly important for governance and auditing.

How to Choose the Right Architecture

The concepts in this chapter should not be applied in isolation. An architectural decision should begin with context.

When designing a new system, several questions can help avoid both fragile architecture and overengineering.

- **What is the current complexity of the domain?** A simple product probably does not need to begin as a distributed system.
- **How many teams will need to modify the system?** Organizational growth may justify stronger boundaries.
- **Which components are likely to change?** Models and providers are natural candidates for isolation.
- **Which parts are security-critical?** Authorization and transactions should have explicit boundaries.
- **Which workloads need to scale independently?** This may justify service separation.
- **What level of operational complexity can the team sustain?** An architecture that looks superior on paper may be inappropriate for the organization's actual capabilities.

A common and healthy strategy is to begin with a well-structured Modular Monolith and extract services only when clear requirements for scale, isolation, or ownership emerge. This path preserves simplicity without blocking future evolution.

Key Trade-Offs in This Chapter

Software Engineering is fundamentally about managing trade-offs. The most important trade-offs in this chapter can be summarized as follows.

- **Abstraction vs. Simplicity:** Abstractions enable evolution, but premature abstractions make systems difficult to understand.
- **Modularity vs. Operational Overhead:** Modules reduce coupling, but excessively distributed components increase complexity.
- **Microservices vs. Monolith:** Microservices increase independence, while monoliths simplify operations.
- **Standardization vs. Team Autonomy:** Standards reduce variability but may constrain experimentation.
- **Maintainability vs. Delivery Speed:** Investing in structure reduces future costs but may reduce immediate delivery velocity.
- **Decoupling vs. Observability Complexity:** Asynchronous systems reduce direct dependencies but make troubleshooting more difficult.
- **Vendor Abstraction vs. Provider-Specific Optimization:** Generic interfaces make migration easier but may prevent the use of provider-specific capabilities.

The role of a Staff Engineer is not to maximize one side of these trade-offs. It is to identify the combination that best fits the organization's stage, risks, and objectives.

Key Cybersecurity Risks Related to Software Design

Software design has a direct impact on the security of AI systems. Several problems frequently emerge when boundaries are poorly defined.

- **LLMs with excessive responsibility:** Models should not directly control authentication, authorization, or critical transactions.
- **Credentials scattered throughout the codebase:** Secrets need to be centralized and protected.
- **Direct trust in external inputs:** User-generated content and documents should be treated as untrusted.

- **Overprivileged components:** Each service should operate with the minimum set of permissions necessary.
- **Lack of validation between components:** Data originating from models or external APIs needs to be validated.
- **Vulnerable dependencies:** SDKs and libraries add attack surface.
- **Inadequate logging:** Logs may accidentally store prompts, PII, or confidential information.

A good architecture reduces the impact of a failure by establishing boundaries and applying independent controls. A compromised model should not automatically mean that the entire infrastructure is compromised. This principle of containment will be essential in the chapters on cybersecurity and agent security.

Business Impact

Software Engineering decisions often appear distant from business metrics, but they have a substantial cumulative economic impact.

A good architecture can produce important outcomes.

- **Lower time-to-market:** Reusable components accelerate the launch of new products.
- **Lower maintenance cost:** Clear boundaries reduce the effort required for changes.
- **Higher reliability:** Fewer failures mean less impact on customers and operations.
- **Lower vendor lock-in:** The organization preserves negotiating power and strategic flexibility.
- **Greater security:** Well-separated architectures reduce the potential impact of incidents.
- **Higher engineering productivity:** Engineers spend less time understanding and fixing fragile structures.
- **Greater scalability:** Systems can grow without requiring frequent rewrites.

This is why architecture should not be treated solely as a technical concern. Architecture is a way of managing the organization's future cost.

Chapter Summary

Software Engineering provides the structures that make it possible to transform AI experiments into sustainable enterprise systems.

The core principles can be condensed into several ideas.

- **Separate responsibilities that change for different reasons:** This reduces coupling and improves security.
- **Build modules around real boundaries:** Modularity preserves the system's ability to evolve.
- **Protect the domain from external details:** Models, databases, and providers should be replaceable whenever possible.
- **Use patterns for recurring problems, not for formalism:** The solution should reduce complexity.
- **Prefer operational simplicity until scale justifies complexity:** Microservices are not a requirement for maturity.
- **Treat maintainability as an economic outcome:** Difficult code costs money.
- **Version behavior, not just code:** Models, prompts, data, and configurations are also part of the system.
- **Design security boundaries explicitly:** Probabilistic logic should never replace critical deterministic controls.

The most important lesson is that a good architecture is not the one with the largest number of patterns or services. It is the one that makes the system easy to understand, modify, operate, secure, and evolve.

Practical Framework for Evaluating the Design of Any AI System

When analyzing an architecture in an interview or in a real-world project, use a set of questions as a guide.

Start with the structure.

- **Responsibilities:** Does each component have a clear responsibility?
- **Boundaries:** Are the boundaries between domain logic, AI, data, and infrastructure explicit?
- **Coupling:** Does a local change require changes across multiple parts of the system?
- **Cohesion:** Does each module have a clear and understandable purpose?
- **Replaceability:** Can models and providers be replaced at a reasonable cost?

Then evaluate operations.

- **Scalability:** Which components need to scale independently?
- **Reliability:** How are failures isolated?
- **Observability:** Can a request be traced end-to-end?
- **Deployment:** Can changes be released and rolled back safely?
- **Maintainability:** Could a new team understand the system?

Next, evaluate security and governance.

- **Trust boundaries:** Where does data move from one trust zone to another?
- **Authorization:** Who actually decides whether an action may be executed?
- **Secrets:** Are credentials isolated?
- **Auditability:** Can the organization reconstruct the system's behavior?
- **Change control:** Are changes to models and prompts also tracked?

Finally, connect architecture to the business.

- **Time-to-market:** Does the structure accelerate or slow down experimentation?
- **Operating cost:** Is operational complexity proportional to the value being produced?
- **Team scalability:** Can new teams use the platform without creating excessive dependencies?
- **Vendor leverage:** Does the organization preserve strategic options?
- **Technical debt:** Does the current architecture increase or reduce future cost?

This mental model represents the behavior expected of a Staff AI Engineer: **use Software Engineering not to produce the most sophisticated architecture, but to control complexity and preserve the organization's ability to continuously generate value.**