A Practical Checklist for Reviewing .NET Application Architecture

A lightweight guide for spotting maintainability risks, unclear boundaries, and codebase decisions that make future change harder.

1 Introduction

Reviewing the architecture of a .NET application is rarely about deciding whether the codebase follows a perfect model. Most long-lived business applications contain compromises, historical decisions, framework influence, urgent fixes, and areas that evolved faster than the original design expected.

The important question is usually more practical:

Can the system still be understood, changed, tested, and operated safely?

This checklist is intended as a lightweight guide for reviewing that question. It focuses on the design signals that often reveal whether a .NET codebase is healthy enough to evolve, or whether maintainability risks are beginning to slow the team down.

The checklist does not assume that every application needs a complex architecture. It also does not treat patterns, layers, abstractions, or frameworks as goals in themselves. A simple application may only need a simple structure. A larger application may need clearer boundaries, stronger separation of responsibilities, and more explicit decisions. The value of an architecture review is not to make the system look more sophisticated, but to make future change safer and easier.

A useful review should therefore look beyond surface structure. It should not only ask whether the solution has projects named Domain, Application, Infrastructure, and Web. It should ask whether those boundaries actually protect decisions. It should not only ask whether dependency injection is used. It should ask whether responsibilities are clear. It should not only ask whether tests exist. It should ask whether the tests explain the behavior and give confidence when the code changes.

This checklist can be used during a codebase review, a technical due diligence exercise, a modernization discussion, or an internal architecture improvement effort. It is especially relevant for existing .NET business applications where the main concern is not starting over, but improving the system without unnecessary rewrites.

The goal is to identify practical next steps:

- where responsibilities have become unclear,
- where framework or infrastructure details have leaked too far inward,
- where business rules are difficult to find or change,
- where tests fail to explain the design,
- where operational concerns such as logging, errors, and cancellation are treated inconsistently,
- and where small improvements could make future changes safer.

The best outcome of a review is not a long list of abstract architecture opinions. It is a clearer understanding of the codebase, the risks that matter most, and a small number of improvements the team can realistically act on.

2 Boundary clarity

Boundaries are useful when they make responsibilities easier to understand. A boundary should not only separate files, folders, namespaces, or projects. It should make it clearer where a decision belongs, which part of the system owns a responsibility, and how one part of the application is allowed to depend on another.

In many .NET applications, boundaries exist visually before they exist architecturally. The solution may contain projects named Web, Application, Domain, and Infrastructure, but the actual responsibilities may still be mixed. Controllers may contain business decisions. Application services may know too much about database details. Domain concepts may be shaped by framework requirements. Infrastructure code may influence how workflows are expressed.

A useful architecture review should therefore ask whether the boundaries protect intent, not only whether they exist.

Purpose

The purpose of reviewing boundary clarity is to understand whether the structure of the application helps the team place new behavior correctly. When a change is needed, the codebase should make it reasonably clear where that change belongs.

A good boundary reduces uncertainty. It helps the team avoid scattering related behavior across unrelated places. It also makes it easier to discuss design decisions without starting from technical implementation details.

What to look for

Look for whether important responsibilities have clear homes.

For example:

- Where are business workflows expressed?
- Where are business rules enforced?
- Where does validation of external input happen?
- Where are persistence details handled?
- Where are framework-specific concerns isolated?
- Where are integrations with external systems placed?
- Where are decisions about authorization, logging, errors, and retries made?

The goal is not to force every concern into a separate project. The goal is to see whether the codebase gives each important responsibility a recognizable place.

Also look at whether the names of projects, namespaces, classes, and methods match the decisions they contain. A boundary is easier to understand when its naming reflects application meaning rather than only technical categories.

Questions to ask

When reviewing boundary clarity, useful questions include:

- Can a developer explain what each project or major namespace is responsible for?
- Does application code express use cases or workflows in language that belongs to the application?
- Are controllers, Razor pages, API endpoints, or UI handlers thin enough that they mainly coordinate input and output?
- Are database, ORM, file system, queue, email, and HTTP details kept away from core application decisions?
- Can business rules be found without searching through UI, persistence, and infrastructure code?
- When a new feature is added, is there an obvious place for the main behavior?
- Do similar decisions tend to be implemented in similar parts of the codebase?
- Are boundaries respected consistently, or only in some areas?

These questions matter because unclear boundaries often create small daily costs. Each change requires more searching, more guessing, and more risk of placing behavior in the wrong part of the system.

Warning signs

Common warning signs include:

- The solution has layers, but responsibilities are still mixed.
- Controllers, page models, handlers, or services contain unrelated decisions.
- Business rules are duplicated across UI, application, and persistence code.
- Infrastructure concerns shape the application model.
- Domain or application code depends directly on framework-specific types.
- New behavior is often added wherever the current developer first finds a convenient place.
- Classes are named after technical activity rather than the decision they represent.
- A change in one workflow often requires edits in many unrelated files.
- It is difficult to explain why a piece of code belongs where it is.

These signs do not automatically mean the architecture is broken. They mean the boundaries may not be carrying enough design responsibility.

Healthy signs

Healthy boundaries usually show up in practical ways:

- Important decisions have clear homes.
- Application workflows can be understood without starting from database or framework details.
- Framework code stays close to the edge of the system.
- Business rules are named and located according to the decisions they represent.
- Similar responsibilities are handled consistently.
- New changes usually have an obvious place to start.
- Tests can exercise application behavior without unnecessary infrastructure setup.

- The team can discuss a change in terms of application behavior before discussing implementation details.

A healthy boundary does not need to be complicated. It simply needs to protect something meaningful.

Review note

When reviewing boundary clarity, avoid judging the application only by its folder structure. A solution can look clean while still hiding unclear responsibilities. The more important question is whether the structure helps the team make better decisions when the system changes.

A useful review outcome might be a short list of boundaries that should be strengthened, simplified, renamed, or made more explicit. In many cases, the best improvement is not to introduce a new layer, but to make an existing responsibility easier to see.

3 Dependency direction

Dependency direction is one of the clearest signals of whether an application's architecture is protecting the right parts of the system. It shows which code is allowed to know about which other code, and which parts of the application are forced to change when technical details change.

In a maintainable .NET application, the most important application decisions should not depend on the most replaceable technical details. Business workflows, policies, rules, and use cases should not be shaped directly by database access, web framework types, UI concerns, queue clients, file systems, email providers, or third-party SDKs.

This does not mean infrastructure is unimportant. It means infrastructure should support application behavior rather than define it.

Purpose

The purpose of reviewing dependency direction is to understand whether the stable parts of the application are protected from volatile details.

A codebase becomes harder to change when important application logic points toward details that are likely to change. If core behavior depends directly on framework APIs, database models, or integration clients, then those details become part of the application's design whether the team intended that or not.

Good dependency direction helps keep the application understandable. It allows the team to describe what the system does before describing how a specific framework, database, or external service makes it happen.

What to look for

Look for which projects, namespaces, and classes reference each other.

Useful things to check include:

- Does application code depend on infrastructure code?
- Does domain or business logic depend on web framework types?

- Do use cases depend directly on Entity Framework, HTTP clients, message queues, or email providers?
- Are abstractions owned by the code that needs them, or by the infrastructure that implements them?
- Are dependencies pointing toward application meaning or toward technical convenience?
- Can important workflows be tested without starting a database, web server, queue, or external service?
- Are external systems represented by application-level interfaces, or do their SDK types leak into core code?

The goal is not to avoid all dependencies. A system without dependencies does not exist. The goal is to make sure dependencies point in a direction that protects the most important decisions.

Questions to ask

When reviewing dependency direction, useful questions include:

- Which parts of the codebase contain the most important business or application decisions?
- Which technical details are most likely to change over time?
- Do stable application concepts depend on volatile implementation details?
- Are abstractions placed near the code that uses them?
- Can infrastructure implementations be replaced without rewriting application workflows?
- Can the team understand a use case without reading persistence or framework code first?
- Are dependency injection registrations hiding design problems rather than solving them?
- Are test doubles simple because the dependencies are well-shaped, or complicated because the design leaks too much detail?

These questions help reveal whether the codebase is organized around application behavior or around the tools used to implement it.

Warning signs

Common warning signs include:

- Domain or application code references infrastructure projects directly.
- Use cases depend directly on Entity Framework `DbContext`, framework request types, or third-party SDK clients.
- Interfaces are created in infrastructure projects and then consumed inward by application code.
- Dependency injection is used heavily, but responsibilities are still unclear.
- Application services pass technical models around instead of application concepts.
- Tests need extensive infrastructure setup to verify ordinary application behavior.
- Replacing a technical dependency would require changes throughout the application layer.
- External service concepts appear in business rule code.
- Core decisions are expressed in terms of database operations, HTTP calls, or framework events.

These signs often indicate that technical details are pulling application code in the wrong direction.

Healthy signs

Healthy dependency direction usually shows up as practical flexibility:

- Application and domain code can describe behavior without depending on infrastructure implementations.
- Infrastructure code depends inward on application-defined contracts.
- Interfaces are named according to the role they play in the application, not only according to the technology used to implement them.
- Framework-specific types stay close to the edge of the system.
- Use cases can be tested with simple substitutes.
- External systems are adapted into application language before their results reach core decisions.
- Database and integration details can change without forcing widespread changes to business workflows.
- Dependency injection connects objects, but the design still makes responsibilities clear without reading the container configuration.

Good dependency direction does not require overengineering. It requires being deliberate about what the application is allowed to know.

Review note

When reviewing dependency direction, do not only look for whether interfaces exist. Interfaces alone do not guarantee good direction. An interface named after a technical dependency can still expose the wrong abstraction if the application code is forced to think in infrastructure terms.

A useful review outcome might identify dependencies that should be inverted, moved closer to the edge, renamed in application language, or hidden behind smaller contracts. Often the goal is not to introduce more abstractions, but to make the existing dependencies point toward the parts of the system that should remain stable.

4 Framework leakage

Frameworks are useful because they handle common technical concerns. ASP.NET Core, Entity Framework, hosting infrastructure, validation libraries, logging frameworks, authentication middleware, serializers, background job tools, and messaging libraries can all remove a large amount of repetitive work.

The problem starts when framework details stop being implementation details and begin shaping the application's own design.

Framework leakage happens when application code is forced to speak too much in the language of a framework. Instead of describing business workflows, decisions, and responsibilities, the code starts to describe controllers, requests, database entities, HTTP responses, ORM behavior, attributes, configuration objects, or SDK-specific models.

Some framework influence is normal. A .NET application must connect to its hosting model, persistence mechanism, UI framework, and integration tools somewhere. The question is whether

those details stay near the edge of the system, or whether they spread into the parts of the code that should express application behavior.

Purpose

The purpose of reviewing framework leakage is to understand whether technical tools are supporting the application or quietly defining it.

A framework should help deliver the application. It should not make important business decisions harder to see. When framework types appear throughout application logic, the team may become dependent on details that are not central to the business problem. This can make the system harder to test, harder to change, and harder to reason about.

Good architecture does not hide frameworks completely. It keeps them in appropriate places.

What to look for

Look for whether framework-specific concepts appear in code that is supposed to express application behavior.

Useful things to check include:

- Do application services depend directly on ASP.NET Core request, response, route, session, or controller types?
- Do business rules depend on Entity Framework `DbContext`, tracked entities, lazy loading, or query behavior?
- Are validation attributes used as the main expression of business rules?
- Do domain or application classes depend on serialization attributes, ORM attributes, or binding-specific requirements?
- Are external SDK models passed deeply into application code?
- Are framework exceptions used to express normal application outcomes?
- Do tests require framework setup to verify ordinary business behavior?
- Are application decisions hidden inside middleware, filters, attributes, or configuration?

The goal is not to remove frameworks. The goal is to see whether framework details are crossing boundaries that should protect application meaning.

Questions to ask

When reviewing framework leakage, useful questions include:

- Can the main application workflows be understood without knowing the web framework?
- Can business rules be explained without referring to database behavior?
- Are framework types limited to adapters, controllers, page models, infrastructure implementations, and composition code?
- Are application concepts represented in application language before they reach core decisions?
- Is the framework helping with transport, persistence, hosting, and integration, or is it shaping the model itself?
- If the framework changed, which parts of the codebase would need to change?
- Are tests focused on behavior, or are they mostly exercises in framework setup?

- Are technical attributes being used where explicit application code would make the decision clearer?

These questions help reveal whether the framework is at the edge of the system or spread through its center.

Warning signs

Common warning signs include:

- Business rules are implemented primarily through framework attributes.
- Application services return framework response types instead of application results.
- Core workflows depend directly on `HttpContext`, controller base classes, page models, or request-specific objects.
- Domain or application models are designed mainly to satisfy ORM or serialization requirements.
- Entity Framework behavior determines how business logic must be written.
- External API or SDK models are passed through many layers unchanged.
- Framework exceptions are used for expected application outcomes.
- Tests need large amounts of framework configuration before they can verify simple behavior.
- Changing a persistence, web, or integration detail would require changes in core application logic.

These signs do not mean the framework is bad. They mean the application may be giving the framework too much design authority.

Healthy signs

Healthy framework use usually has these characteristics:

- Framework-specific code is easy to find and mostly stays near the application boundary.
- Controllers, page models, endpoints, middleware, and infrastructure classes translate between framework concerns and application concerns.
- Application services use application-specific inputs and outputs.
- Business rules are expressed in explicit code, with names that describe the decision being made.
- Persistence details are handled by repositories, query services, data access components, or infrastructure adapters.
- External service models are translated before they influence core decisions.
- Tests for application behavior can run without starting the full framework stack.
- Framework configuration supports the design instead of hiding it.

In a healthy codebase, the team can still benefit from framework productivity without letting the framework dominate the application language.

Review note

When reviewing framework leakage, avoid turning the review into a general criticism of frameworks. The issue is not whether ASP.NET Core, Entity Framework, or any other tool is being used. The issue is whether those tools have been allowed to cross into places where application decisions should remain explicit.

A useful review outcome might identify framework dependencies that should be moved outward, translated earlier, wrapped behind application-specific contracts, or replaced with clearer application results. Often the best improvement is small: move the framework detail closer to the edge, give the application decision a better name, and make the code express the behavior rather than the tool.

5 Business rules and validation

Business rules and validation are often discussed together, but they do not represent the same kind of decision.

Validation is usually about whether uncertain input is acceptable enough to enter the system. Business rules are about whether a requested action is allowed according to the meaning of the application. Guard clauses are usually about protecting code from invalid internal use or impossible states.

The distinction matters because different kinds of checks belong in different places and should fail in different ways. When every check is treated the same, the codebase can become harder to understand. Expected user mistakes may be handled like programming errors. Business decisions may be hidden inside generic validation code. Internal guard clauses may be used to represent ordinary application outcomes.

A maintainable application should make these differences visible.

Purpose

The purpose of reviewing business rules and validation is to understand whether the codebase clearly separates input quality, internal correctness, and application decisions.

A useful review should ask whether checks are named according to the decision they represent. It should also ask whether failure is communicated in a way that matches the situation.

Some failures should produce validation messages. Some should produce application results. Some should throw exceptions because they represent bugs or invalid internal use. Treating all of these as the same kind of failure makes the system less expressive and often harder to test.

What to look for

Look for where different kinds of checks are performed.

Useful things to check include:

- Where is external input validated?
- Where are business rules enforced?
- Where are guard clauses used?
- Are ordinary user mistakes handled differently from programming errors?
- Are business decisions named explicitly, or hidden inside generic validation methods?
- Are exceptions used for expected application outcomes?
- Are result types used where the caller needs to respond to a normal business outcome?
- Are validation rules placed close to the boundary where uncertain input enters the system?
- Are rules duplicated between UI, application services, domain code, and persistence code?

The goal is not to create a complicated taxonomy. The goal is to make the code communicate what kind of decision is being made.

Questions to ask

When reviewing business rules and validation, useful questions include:

- Is this check about external input, internal correctness, or an application decision?
- Should failure be expected as part of normal use?
- Should the caller be able to respond to the failure?
- Is an exception being used for something the application expects can happen?
- Is a validation mechanism being used to hide a business rule?
- Are guard clauses protecting internal assumptions, or are they controlling user-facing behavior?
- Are rule names meaningful enough that tests can explain the behavior?
- Can a developer find the rule without knowing a specific UI, database, or framework detail?
- Are similar failures represented consistently across the application?

These questions help reveal whether the code is expressing the decision clearly, or only enforcing it somewhere.

Warning signs

Common warning signs include:

- Business rules are implemented as generic validation attributes.
- Expected business outcomes are represented by exceptions.
- Guard clauses are used for normal user-facing decisions.
- Validation logic is scattered across UI, application, domain, and persistence code.
- The same rule is repeated in several places with slightly different behavior.
- Rule names describe technical checks rather than business decisions.
- Tests focus on implementation details instead of the meaning of the rule.
- It is difficult to tell whether a failure means bad input, a forbidden action, or a programming error.
- Error messages, result types, and exceptions are used inconsistently for similar situations.

These signs often indicate that the codebase enforces rules without clearly explaining them.

Healthy signs

Healthy handling of rules and validation usually looks like this:

- External input is validated close to the boundary where it enters the system.
- Business rules are named according to the decision they represent.
- Guard clauses protect internal assumptions and fail fast when code is used incorrectly.
- Expected business outcomes are returned as explicit results when the caller needs to respond.
- Exceptions are reserved for exceptional situations, bugs, or unexpected failures.
- Tests describe the rule in application language.
- Similar decisions fail in similar ways.
- Business rules can be found without searching through framework-specific validation code.

- The code makes it clear whether a failure is a validation problem, a business decision, or an internal correctness issue.

A healthy codebase does not necessarily have a separate rule engine or complex validation framework. It simply makes important decisions visible and gives each kind of failure an appropriate shape.

Review note

When reviewing business rules and validation, avoid asking only whether the application "has validation." Almost every application validates something. The more important question is whether each check is placed, named, and represented according to the decision it makes.

A useful review outcome might identify rules that should be moved out of generic validation code, expected failures that should become result types, guard clauses that should be limited to internal correctness, or duplicated rules that should be expressed once in clearer application language.

The best improvements are often small. Rename a rule so the decision is visible. Move validation closer to the boundary. Replace an expected exception with an explicit result. Make a test describe the rule rather than the implementation detail.

6 Testing as design feedback

Tests are often treated mainly as a safety net. That is important, but it is not the only value they provide. Tests also reveal how easy or difficult the application is to use, understand, and change.

When tests are hard to write, hard to read, or hard to maintain, they may be exposing design problems in the production code. Complicated setup can reveal unclear dependencies. Fragile assertions can reveal behavior that is not well named. Overly clever test doubles can reveal contracts that are too broad or too vague. Tests that focus only on technical details may reveal that the application behavior itself is difficult to express.

A useful architecture review should therefore look at tests not only as quality assurance, but also as design feedback.

Purpose

The purpose of reviewing tests as design feedback is to understand whether the codebase can explain its behavior through tests.

Good tests should make important application behavior visible. They should help a reader understand what the system is expected to do, why a decision matters, and what kind of change would break that expectation.

Tests do not need to be perfect. They do not need to cover every line. But the most important behavior should be represented in a form that gives the team confidence when the code changes.

What to look for

Look for whether the tests explain application behavior or mostly reproduce implementation details.

Useful things to check include:

- Do test names describe behavior in application language?
- Can important use cases be tested without excessive setup?
- Are test doubles simple and explicit?
- Do tests make business rules, validation decisions, and workflow outcomes visible?
- Are assertions focused on observable behavior rather than incidental implementation details?
- Do tests fail for meaningful reasons?
- Are tests arranged so that the scenario is easy to understand?
- Are integration tests used where infrastructure behavior matters?
- Are unit tests used where application decisions can be tested without infrastructure?
- Are tests maintained as part of the design, or treated as a separate afterthought?

The goal is not to force one testing style everywhere. The goal is to see whether the test suite helps the team reason about the application.

Questions to ask

When reviewing tests as design feedback, useful questions include:

- What does this test teach a reader about the system?
- Does the test name reveal the decision being protected?
- Is the setup proportional to the behavior being tested?
- Are test doubles boring and explicit, or are they hiding too much behavior?
- Does the test verify an outcome that matters to the application?
- Would this test still make sense if the implementation changed?
- Are important rules tested where they are expressed?
- Are tests grouped around behavior, workflows, or decisions, rather than only around classes?
- When a test fails, does the failure help identify what behavior changed?

These questions help distinguish tests that support design from tests that merely exercise code.

Warning signs

Common warning signs include:

- Tests require large amounts of setup before the actual behavior appears.
- Test names describe method calls instead of expected behavior.
- Test doubles contain complicated logic that competes with the production code.
- Many tests break when harmless implementation details change.
- Assertions focus on internal calls rather than meaningful outcomes.
- Important business rules are only tested indirectly through UI or database-heavy tests.
- Tests are difficult to read without already knowing the implementation.
- Similar scenarios are tested inconsistently across the codebase.
- The team avoids refactoring because the tests are too fragile or too unclear.

These signs often indicate that the tests are not giving the team enough design feedback.

Healthy signs

Healthy tests usually have these characteristics:

- Test names describe behavior, decisions, or scenarios in application language.

- Setup is small enough that the important behavior is visible.
- Test doubles are simple, explicit, and close to the behavior under test.
- Assertions focus on observable outcomes.
- Important business rules have direct tests.
- Infrastructure-heavy tests are reserved for behavior that actually depends on infrastructure.
- Tests can fail in ways that help the team understand what changed.
- The test suite gives confidence to make small improvements.
- Tests make the design easier to explain to a new developer.

Healthy tests do not need to be clever. In many cases, the best tests are deliberately boring because they make the scenario, rule, and expected result obvious.

Review note

When reviewing tests, avoid treating coverage numbers as the only measure of quality. Coverage can show which code was executed, but it does not show whether the tests explain the system well.

A useful review outcome might identify areas where tests should be renamed, simplified, moved closer to the behavior they protect, or rewritten to focus on application outcomes. It might also reveal production code that should be reshaped because ordinary behavior is too difficult to test clearly.

The best testing improvements often come from asking a simple question: what should this test help the next developer understand?

7 Operational concerns

Architecture is not only about how code is organized. It is also about whether the application can be understood and supported when it is running.

A system may have reasonable boundaries, good dependency direction, clear business rules, and useful tests, but still be difficult to operate. If errors are hard to diagnose, logs do not explain decisions, cancellation is inconsistent, background work fails silently, or external failures are handled differently in different parts of the system, the application can become risky in production even if the code looks clean.

Operational concerns are design concerns because they affect how safely the system behaves outside the development environment.

Purpose

The purpose of reviewing operational concerns is to understand whether the application provides enough information and control when things happen in production.

A useful review should ask whether the system helps developers and operators answer practical questions:

- What happened?
- Why did it happen?
- Which user, request, job, or integration was involved?
- Was the failure expected or unexpected?
- Can the operation be retried safely?

- Did the system stop, continue, compensate, or ignore the problem?

If the application cannot answer these questions, production issues become harder to investigate and changes become harder to trust.

What to look for

Look for whether operational behavior is handled deliberately and consistently.

Useful things to check include:

- Are logs written at meaningful decision points?
- Do log messages explain decisions, not only exceptions?
- Are errors represented consistently across the application?
- Are expected failures separated from unexpected failures?
- Is cancellation supported where operations may be long-running or user-initiated?
- Are `CancellationToken` values passed through public async workflows?
- Are retries, timeouts, and fallback behavior explicit where external systems are involved?
- Are background jobs and scheduled tasks observable?
- Are integration failures logged with enough context to diagnose them?
- Are sensitive values kept out of logs and error messages?
- Are important operations traceable across application, infrastructure, and external boundaries?

The goal is not to add logging everywhere. The goal is to make important behavior explainable when the system is running.

Questions to ask

When reviewing operational concerns, useful questions include:

- If this operation fails in production, how would the team know?
- Would the logs explain what decision was made, or only that an exception occurred?
- Are expected business failures logged differently from system failures?
- Are external service failures handled in a consistent way?
- Are timeouts and retries explicit, or hidden inside technical defaults?
- Can long-running operations be cancelled safely?
- Are cancellation tokens passed through the call chain, or accepted and then ignored?
- Are background processes visible enough to diagnose missed or failed work?
- Does the code make it clear whether retrying an operation is safe?
- Are identifiers included that make an issue traceable without exposing sensitive data?

These questions help reveal whether operational behavior is part of the design or only added after problems occur.

Warning signs

Common warning signs include:

- Logs mostly say that something failed, but not why the application made a decision.
- Exceptions are caught and ignored.
- Expected failures and unexpected failures are logged in the same way.

- Different parts of the application handle external failures differently.
- CancellationToken parameters are accepted but not passed further down the call chain.
- Async workflows cannot be cancelled even when the caller has gone away.
- Retry behavior is scattered, duplicated, or implicit.
- Background jobs fail without clear visibility.
- Error messages expose technical details or sensitive information.
- Production diagnosis depends on reproducing the issue locally.
- Important workflows cannot be traced across boundaries.

These signs often indicate that operational behavior is accidental rather than designed.

Healthy signs

Healthy operational design usually has these characteristics:

- Logs are placed at meaningful decision points.
- Log messages explain what happened and why it matters.
- Expected failures are represented and handled differently from unexpected failures.
- External calls have deliberate timeout, retry, and failure-handling behavior.
- Cancellation is part of the public async contract where relevant.
- Background work is observable.
- Important operations include enough context to diagnose issues safely.
- Sensitive data is not written to logs.
- Error handling is consistent across similar workflows.
- The team can investigate production problems without guessing through unrelated code.

Healthy operational behavior does not require a large observability platform from day one. It requires that the application's important decisions and failures are visible enough to support real use.

Review note

When reviewing operational concerns, avoid treating logging, error handling, and cancellation as separate technical chores. They are part of the application's behavior.

A useful review outcome might identify workflows where logging should explain decisions more clearly, exceptions should be classified more consistently, cancellation should be passed through properly, or external calls should have explicit timeout and retry behavior.

The best improvements are often small but valuable: add a meaningful log message at a decision point, pass a cancellation token through the call chain, distinguish an expected business result from an unexpected failure, or make a background operation visible enough to diagnose when it goes wrong.

8 Review summary checklist

The previous sections describe signals to look for when reviewing an existing .NET application. This final checklist is intended as a compact summary that can be used during or after a review.

It is not meant to produce a perfect architecture score. The purpose is to identify the areas that most affect maintainability, change safety, and the team's ability to understand the system.

For each area, mark the risk level as L, M, or H — Low, Medium, or High — then add short notes about what was observed and what should be improved.

Risk levels

Low risk means the area is generally clear, consistent, and unlikely to block ordinary change.

Medium risk means the area works, but there are signs of drift, duplication, unclear responsibility, or inconsistent practice.

High risk means the area regularly makes change harder, hides important decisions, spreads responsibility across unrelated code, or creates avoidable production or maintenance risk.

Summary table

Area	What to check	Risk level	Notes
Boundary clarity	Important responsibilities have clear homes, and the structure helps developers understand where changes belong.	L / M / H	
Dependency direction	Stable application decisions are protected from volatile framework, database, infrastructure, and integration details.	L / M / H	
Framework leakage	Framework-specific types and concepts stay close to the edge instead of shaping core application behavior.	L / M / H	
Business rules and validation	Validation, guard clauses, and business rules are placed, named, and represented according to the decision they make.	L / M / H	
Testing as design feedback	Tests explain application behavior, protect important decisions, and give confidence when the code changes.	L / M / H	
Operational concerns	Logging, error handling, cancellation, retries, and background work make production behavior understandable and supportable.	L / M / H	

Risk level abbreviations: L = Low, M = Medium, H = High.

Follow-up questions

After completing the table, the most useful questions are usually:

- Which risk appears most likely to slow the team down in the next few months?
- Which issue makes ordinary changes harder than they should be?
- Which improvement would make the next change safer?
- Which problem is repeated across several areas?
- Which issue can be improved incrementally without a large rewrite?
- Which decision is currently hidden but should be made explicit?

- Which improvement would help the team understand the codebase better?

The answers to these questions are often more valuable than the risk labels themselves.

Prioritizing improvements

A useful review should not produce a long list of unrelated recommendations. It should identify a small number of improvements that are worth acting on.

A practical prioritization might use three groups:

Act soon

These are issues that create frequent change risk, recurring defects, unclear ownership, or production support problems. They should be addressed because they affect the team's ability to work safely.

Improve gradually

These are areas where the design is not ideal, but the risk can be reduced through normal feature work, small refactorings, better tests, or clearer naming. They do not require a separate rewrite effort.

Leave alone for now

These are imperfections that are visible but not currently expensive. Not every architectural compromise deserves immediate action. Sometimes the best decision is to document the tradeoff and avoid unnecessary churn.

Review outcome

The final review outcome should be short enough to act on.

A useful summary might include:

- the strongest parts of the current design,
- the main maintainability risks,
- the decisions that are hardest to see,
- the areas where framework or infrastructure details have leaked too far,
- the tests or operational signals that are missing,
- and three to five practical next steps.

The goal is not to make the application look architecturally perfect. The goal is to help the team understand where future change is likely to become difficult, and what can be improved without losing sight of the real business context.

A good architecture review should leave the team with clearer language, fewer vague concerns, and a small set of improvements that can actually be started.