Complex C# Patterns That Confuse LLMs in Unit Test Generation

Large Language Models often struggle to generate correct unit tests for **ASP.NET Core** or **Clean Architecture**-style C# applications that use sophisticated design patterns. Below we explore several patterns and architectural choices that consistently mislead LLMs (like o3-mini or even GPT-based models) when asked to create unit tests with **Moq** and **xUnit/NUnit**, along with the common mistakes and reasoning errors observed.

Heavy Dependency Injection and Multiple Service Layers

Problem: In "clean" ASP.NET Core architectures, classes often depend on many injected services (sometimes dozens). To unit test such a class, one must provide *all* required dependencies – usually as mocks – which is cumbersome and non-intuitive for an LLM. The model may omit certain dependencies or incorrectly instantiate the class under test.

Example: Consider a service FooLogicImplementer that depends on an **enumerable of interfaces** (e.g. IEnumerable<IFooCollection>) – essentially multiple service layers feeding into one logic classabhinavcreed13.github.ioabhinavcreed13.github.io. In a real unit test, a developer must create *multiple* mocks (for each IFooCollection and their inner IFooWork objects) and assemble them into a list before instantiating the target classabhinavcreed13.github.ioabhinavcreed13.github.io. An LLM often fails to reason through this setup. It might:

- **Omit Dependencies:** Forget to provide one of the constructor parameters or assume defaults exist (leading to compilation errors).
- **Improper Instantiation:** Try to call a nonexistent parameterless constructor or use an DI container in the test (which isn't how unit tests supply dependencies).
- **Superficial Mocks:** Create mocks but not use them correctly. For instance, an LLM might declare mocks but then never inject them, resulting in a NullReferenceException or no interactions at runtime – a common error observed in generated testsarxiv.org.

Example dependency graph: A logic class depends on multiple interfaces (IFooCollection containing IFooWork instances). Testing this requires mocking each interface and assembling them appropriately – a setup LLMs often mishandle.

LLM Mistakes: Missing or improperly using mocks is a frequent failure mode. A study of LLM-generated tests found that *"improper or missing mocking"* of required inputs was a top cause of test failuresarxiv.org. For example, an AI might not realize it needs to mock a repository or an external service, leading to tests that abruptly terminate or always fail due to unprovided dependencies[arxiv.org]. Another observed mistake is creating the mocks but never verifying or using them (e.g. expecting a call was made, but the LLM didn't actually set up the invocation), resulting in *"zero interactions with mocks"* being recorded[arxiv.org].

Complex Interfaces and Polymorphism

Problem: Enterprise C# code uses interfaces, base classes, and polymorphic behavior extensively (e.g. strategy patterns, multiple implementations of one interface). An LLM can be confused about which concrete types to use or how to configure mocks for polymorphic behavior. This is especially tricky if the code's behavior depends on the actual runtime type or if extension methods are involved in interface usage.

Example: A common pattern is a *strategy resolver* that picks an implementation based on input. Suppose an interface IPaymentStrategy has multiple implementations, selected via a factory method. Writing a test requires understanding which implementation will be chosen and setting up the factory or DI container accordingly. LLMs often lack this context. In one case, a developer had to unit test a **ServiceProvider extension method** that resolves a strategy; they noted the "challenge" was having to **mock a .NET framework class and replicate extension method behavior**[medium.com]. An LLM not aware of Moq's limitations with extension methods might attempt to call the extension directly or ignore the need to simulate the IServiceProvider properly.

Common LLM Mistakes:

- **Using Wrong Types:** The LLM might instantiate or refer to a concrete class that isn't actually used in the system under test (e.g. calling methods on a base class instead of the interface). This can lead to *hallucinated method calls* or simply ineffective tests that don't trigger the real logic.
- **Not Mocking the Right Layer:** When polymorphism is involved, the AI sometimes mocks the wrong thing. For example, if testing a repository that internally calls an IDatabase interface, an inexperienced LLM might try to mock the repository itself (which defeats the purpose) instead of the lower IDatabase dependency (a mistake even human juniors make). A Reddit example discussing Moq highlights *"mocking the wrong thing"* as a core issue in such scenarios[reddit.com].
- **Ignoring Context for Interfaces:** If a method's outcome depends on which implementation of an interface is provided, a generated test might not supply the correct implementation or any at all, thereby missing the code path it should verify. The result is often an assertion of a default or null outcome, because the LLM didn't set up the context required for the polymorphic behavior to manifest.

Asynchronous Workflows (async/await and Task)

Problem: Async methods and Task-returning interfaces introduce timing and setup complexities. Tests need to await calls, and mocks should use ReturnsAsync or return completed Tasks. LLM-generated tests frequently make mistakes in handling async, such as forgetting to await or using the wrong Moq setup, leading to compile errors or tests that pass trivially without actually testing the async behavior.

Example: In a service with an async method Task<Foo> DoWorkAsync(), a proper Moq setup might be: mock.Setup(x => x.DoWorkAsync()).ReturnsAsync(expectedFoo);. An LLM might instead do Returns(expectedFoo) (omitting the Task wrapper) or neglect the async altogether. In the earlier FooLogicImplementer test, the developer uses Task.FromResult(...) to return a completed task from the mockabhinavcreed13.github.io. This is a nuanced detail – Moq also offers .ReturnsAsync(...) – but an AI often misses it. It might produce a non-awaitable setup or call the async method without awaiting it. These mistakes cause either compilation issues or logic that doesn't actually wait for the result (possibly leading to false passes).

Common LLM Mistakes:

- **Forgetting await:** The model might call an async method as if it were synchronous. This can lead to tests that always pass (because they never actually execute the method's contents) or that incorrectly ignore exceptions thrown inside the task. It's essentially an *incomplete Act step* – the test finishes before the system under test does its work.
- **Mismatched Setup:** When configuring mocks, LLMs sometimes use the wrong return type. For example, trying to return a raw value where a Task<T> is expected. This is an *API misuse* – akin to the "argument type mismatches" seen as common LLM coding errorsarxiv.org. The code may not compile at all, or if it does (in case of Returns(Task.FromResult(...)) vs. ReturnsAsync confusion), it might not behave as intended.
- **Async Void or Fire-and-Forget:** If the code uses event handlers or fire-and-forget tasks (which return void or are not awaited), an AI might not know how to test that. It could either ignore the async event or insert a naive delay (Thread.Sleep) to "wait" for it – a practice which can lead to flaky tests. Indeed, testing asynchronous timing with sleeps is brittle: *"a test that tends to fail"* unpredictablyhelpercode.comhelpercode.com. LLMs lacking deeper insight might resort to exactly such brittle approaches for waiting on background tasks.

Event-Driven Systems (Events, IObservable, Domain Events)

Problem: Event-driven architectures (such as using C# events, Reactive IObservable<T> streams, or domain event dispatchers) are hard to test because the effects are indirect. The test must subscribe to events or otherwise intercept them. LLMs often completely miss how to handle events in unit tests, either by ignoring event verification or by making incorrect assumptions about their firing.

Example: Suppose a method triggers an event when something happens (e.g. OnOrderPlaced). A proper test would attach a handler or use Moq's .Raise to simulate the event. LLMs often fail here. They might call the method and then attempt to assert that "event was raised" without any mechanism to catch it, essentially making a meaningless assertion. Moq does support raising events via mocks, but it requires setting up the event and invoking .Raise(...) on the mock when appropriate. If the event is tied to an async method, it gets even trickier – as one Stack Overflow note points out: *"With an async method on a mock, you need to first specify that it returns a Task **before** you can have it trigger events."*stackoverflow.com. That's a subtle ordering that an AI is likely to overlook.

Common LLM Mistakes:

- **No Subscription or Capture:** The LLM might not subscribe to an event at all, yet still assert that some outcome of the event occurred. For example, it may call a method that internally raises an event and then assert that a flag is true, without ever listening for the event to set that flag in the test context.
- **Incorrect Event Simulation:** Some AI-generated tests try to manually call event handlers or simulate events in nonsensical ways. Without understanding the event hookup, an LLM might do something like myObject.OnEvent(null, EventArgs.Empty) (directly invoking the event method) instead of using the proper pattern (like mock.SomeEvent += handler; ... mock.Raise(...) in Moq). This stems from a reasoning error about how events work in C#.
- **Missing Async Event Handling:** As mentioned, if the event or observable uses async (like an IObservable<T> that schedules work), the AI might not wait for the event to propagate. It could conclude the test before the event processing is complete, leading to false negatives (test failing because it didn't wait) or false positives (test passes without actually verifying the event logic).

Testing reactive streams (Rx) or domain events often requires time-based testing (using frameworks or virtual schedulers) and understanding of hot vs cold observables. LLMs have difficulty with such concepts, frequently leading to tests that either ignore the reactive aspect or attempt to assert on internal state changes that are not exposed.

Use of Factories, Strategies, and Other OOP Patterns

Problem: Advanced object-oriented patterns like Factory Method, Abstract Factory, Strategy, etc., introduce indirection. The code under test may not directly instantiate its collaborators; it asks a factory. Or it chooses a strategy at runtime. LLMs often get “lost” following these indirections, resulting in tests that set up nothing useful or attempt to new-up objects that should be obtained via factories.

Example: In a Factory pattern, if a class calls var widget = _widgetFactory.Create(type), a test must inject a mock _widgetFactory that returns a fake widget. LLMs might instead try to instantiate a concrete Widget directly, which bypasses the factory logic entirely (thereby not actually testing the factory-handling code). We saw a real example with an extension method acting as a factory/locator for strategies: the author had to *“replicate the behavior to mock it”* because it was an extension on a system interfacemedium.com. This kind of setup (e.g. faking what an extension method would do, or injecting a specific strategy implementation) is far from straightforward, and an AI is likely to either skip it or mishandle it.

Common LLM Mistakes:

- **Bypassing the Pattern:** The AI might inadvertently short-circuit the very pattern you’re testing. For instance, if the code is supposed to use the **Strategy pattern** to choose algorithm A or B, an ill-conceived test might directly call algorithm A’s class, not exercising the strategy selection logic at all.
- **Mocking the Wrong Thing (Again):** Factories can confuse LLMs as to what should be mocked. Should it mock the factory so it returns a dummy product? Or should it mock the product’s behavior? Often the right approach is to stub the factory. GPT-like models might instead try to mock the product methods, leaving the factory call intact (and thus still creating a real object, defeating isolation). This indicates a reasoning gap in tracing dependencies.
- **Instantiation Errors:** Some LLM outputs have been known to “hallucinate” methods or constructors (e.g., calling a non-existent new SpecificStrategy(type) because it vaguely remembers a pattern). These *hallucinated API calls*arxiv.org lead to tests that won’t compile. In complex pattern-heavy code, if there’s anything not explicitly shown to the LLM, it may invent helper methods to bridge the gap – a clear failure in following the actual code structure.

Edge Cases in Middleware and Exception Handling

Problem: ASP.NET Core middleware, filters, and configuration-based branches pose a challenge. They often rely on runtime configuration (HTTP context, environment settings) or only throw exceptions under specific conditions. LLMs, lacking true runtime understanding, frequently overlook how to simulate those conditions in a test.

Example: Consider a custom ASP.NET Core middleware that reads a header and throws an exception if a value is missing, but only in Production environment. An LLM might generate a test that simply calls the middleware's Invoke method without setting up an HttpContext or required headers – obviously, that test will fail or not even compile. The correct approach would be to construct a DefaultHttpContext, perhaps use a TestServer or at least pass in a dummy RequestDelegate to the middleware. These are *advanced techniques* documented in Microsoft docs for middleware testinglearn.microsoft.com, but unlikely to be reproduced correctly by a language model without explicit hints.

For exception handling paths, such as a domain service that only throws under certain config values, the AI might not realize it needs to simulate that config. For example, if a service method checks if (config.IsFeatureXEnabled) throw ..., an AI-generated test could erroneously expect an exception unconditionally, or never assert the exception at all, missing that branch entirely.

Common LLM Mistakes:

- **No HttpContext or Pipeline Setup:** When testing middleware or controllers, failing to provide the necessary context objects. The model might call await middleware.Invoke(null!) or similar nonsense, not understanding the need for a valid HttpContext. This results in runtime null exceptions or tests that don't actually test the middleware logic.
- **Assuming Defaults:** If code behaves differently in development vs production (via IConfiguration or IHostEnvironment checks), the LLM might assume one and not make it configurable in the test. This can mean the test never triggers the branch it's meant to (e.g., it always passes because the exception branch wasn't activated).
- **Catching/Not Catching Exceptions Incorrectly:** Another error is the AI either forgetting to assert that an exception is thrown, or catching it incorrectly. Sometimes GPT-style output will surround code in try-catch and then Assert.Fail() on exception, which is the opposite of what you want when the exception is the expected behavior. This stems from a lack of clarity on whether the exception is intentional or signals a test failure.

In general, code that depends on external context (HTTP requests, config, environment variables, etc.) requires extra setup in tests (like using Environment.SetEnvironmentVariable or custom IOptions<T> injection). Those nuances are frequently absent in one-shot AI-generated tests, causing them to either be incomplete or flat-out wrong about the code's behavior.

Unobservable Side Effects (Logging, Console, State Mutation)

Problem: Functions that primarily produce side effects – e.g. logging, writing to console, updating internal state without returning a value – are hard for an LLM to verify. Since these side effects aren't directly exposed, tests need to use indirect means (such as intercepting logs or querying internal state via reflection or hooks). AI models often either ignore verifying these effects or attempt to assert things that aren't actually accessible.

Example: A method DoWork() logs an error and returns void. A real unit test might use a mock ILogger<T> and verify that LogError was called with the expected message. However, verifying logger calls is tricky because of extension methods on ILogger (like LogInformation, LogError are extension methods on the interface). The tester must verify the underlying ILogger.Log was called with correct parametersdamirscorner.comdamirscorner.com. An LLM is unlikely to remember this. If asked to "ensure it logs an error," it might do nothing (skipping the assert), or it could hallucinate a property like logger.LastMessage to check – not a real thing.

Common LLM Mistakes:

- **No Assertion for Side Effect:** The test may just call DoWork() and succeed without asserting that the side effect happened. This yields a test with *zero value* – it doesn't actually validate behavior, a failure of reasoning where the model didn't know how to observe the effect.
- **Incorrect Hooking:** If it does try to verify, the AI might misuse the framework. For example, attempting loggerMock.Verify(l => l.LogInformation("Expected message")) directly. Because LogInformation is an extension, this won't catch anything. The correct approach is verifying the internal Log(LogLevel, ...) calldamirscorner.comdamirscorner.com. Many LLMs are unaware of this and so produce assertions that always pass (never actually checking the real call) or won't compile.
- **Global State Assumptions:** For side-effects like writing to Console or modifying a static field, an LLM might assert that "console output contains X" without capturing the console output stream in the test. Or it might assume the static state can be queried in test, which it often cannot (if not exposed). This shows a lack of understanding of how to observe indirect effects. In practice, developers use dependency injection for such concerns (e.g. injecting an IOutput or using patterns to avoid static writes), but an AI won't introduce a new seam to make testing easier – it only has the given code, so it might make an *impossible assertion*.

In summary, unobservable effects often lead to AI tests that are either no-ops or incorrect. Real-world testing techniques (like using in-memory log providers, or exposing internal state via friend assemblies) are outside the knowledge of most LLMs unless specifically trained on those exact patterns.

Concurrency and Race Conditions

Problem: Concurrency issues (threading, race conditions, locks) are notoriously hard to test even for humans. They often require stress testing or specialized synchronization in tests. LLMs completely lack the ability to reason about nondeterministic interleavings, so they either ignore concurrency or attempt overly simplistic tests.

Example: Imagine a method that starts two threads to increment a counter and is supposed to be thread-safe. A proper test might try to run that method many times in parallel and then assert the final count, or use synchronization primitives to create a specific interleaving. An AI, on the other hand, might just call the method once and assert the counter increased – missing the point of the concurrency test. In some cases, we've seen AI-generated tests insert Thread.Sleep calls "to wait" for threads to finish, as a naive way to deal with asynchrony. This is brittle and not reliable: *"Writing unit tests for multi-threaded code is not simple and could even be impossible for some scenarios"*helpercode.com, and adding arbitrary delays often leads to flakinesshelpercode.com.

Common LLM Mistakes:

- **Deterministic Assumptions:** The LLM might write the test as if the multi-threaded code were sequential. For example, if two tasks race to update a value, the AI could assume one finishes first and assert on that order – which may sometimes fail. It doesn't grasp the nondeterministic nature, so it cannot generate proper assertions that allow for timing variance.
- **Lack of Loop or Stress Testing:** A human might run a loop of 1000 iterations to probabilistically catch a race condition. An AI usually won't spontaneously do that (unless instructed), because it doesn't inherently strategize about increasing race odds. It might just do one iteration, which likely won't catch the bug. In effect, the test generated doesn't actually *test* for the race condition.
- **Ignoring Thread Safety**: If the code under test requires synchronization, an AI might not realize it needs to assert that no data race occurred. Since that is hard to assert directly, one would assert final states or invariants. The AI often lacks the concept of an invariant to check. For example, for a producer-consumer queue, a good test might ensure no items are lost or duplicated under concurrent access. An AI might simply enqueue and dequeue on a single thread, thus not testing concurrency at all.

In general, LLMs don't possess the dynamic analysis ability to handle multi-threaded execution. They can't truly run the code to observe a race, so unless the prompt explicitly describes the concurrency issue, they will likely miss it. Concurrency tests frequently require custom tooling (like Microsoft's ConcurrencyVisualizer or tools like Holodeck in tests) – far beyond what an LLM can conjecture. As a result, AI-generated tests either skip concurrency concerns or produce nonsensical attempts (like extremely long sleeps, or expectations that sometimes randomly fail).

Conclusion and Key Takeaways

In summary, certain **design patterns and coding practices in C# make it very challenging for LLMs to generate correct unit tests**. These include heavy use of DI, layered architectures, polymorphic designs, asynchronous and event-driven flows, hidden configuration toggles, side-effect-heavy methods, and concurrent code. The **common failure theme** is that the AI lacks contextual understanding of the code's execution environment and contract. It often produces tests that are superficially plausible but flawed in reasoning – such as missing necessary setup, asserting the wrong things, or ignoring important outcomes.

Developers have noted that current LLMs *"are not so advanced to generate unit tests from scratch"* for complex scenarios – choosing what and how to mock is a particularly hard problemmedium.com. Empirical evaluations back this up: even state-of-the-art models produce tests with many compilation and logic errors that require manual correctionarxiv.orgarxiv.org. Some of the most frequent mistakes include improper use of mocking frameworks, hallucinated method calls, missing imports or context, and incorrect expected values due to weak logical reasoningarxiv.org.

When dealing with **Clean Architecture** or similarly abstracted designs, it's crucial to verify AI-generated tests carefully. They tend to **over-simplify** the scenario, often testing only the "happy path" and forgetting edge cases or setup needed for unhappy paths. As we've seen, things like logger verification or event raising are subtle – even human-written tests consider them trickydamirscorner.com – and LLMs will stumble without explicit guidance.

Bottom line: The more indirection, abstraction, or non-linear control flow in your code, the more likely an AI will misinterpret how to test it. LLMs excel at boilerplate test structure, but **fall short on intricate reasoning** about dependency behaviors, async timing, or hidden side effects. Knowing these weak spots, developers can anticipate which generated tests need extra scrutiny or augmentation. Until LLMs gain a deeper semantic understanding of code execution, complex C# application patterns will continue to produce AI-written tests that require significant fixes to be usefulmedium.comarxiv.org.

Why LLMs Struggle with Complex C# ASP.NET Core Architectures in Unit Test Generation

Dependency Injection and Clean Architecture Complexities

Modern ASP.NET Core applications often follow *Clean Architecture* principles with strict layering and heavy use of dependency injection. In such systems, business logic is decoupled behind numerous interfaces and services. **LLMs frequently falter in this scenario**, as they must identify and provide test doubles for each dependency. In practice, ChatGPT and similar models often omit or mis-handle certain dependencies in generated tests, leading to compile-time errors (e.g. undefined interfaces or classes)mingwei-liu.github.io. Community feedback echoes this – one developer noted that ChatGPT would default to using Moq for mocks even when the project used a different framework (NSubstitute), requiring very specific prompting to correct itreddit.com. These issues suggest that complex IoC/DI configurations exceed the model's one-shot reasoning; without *deep knowledge of the code base's structure*, the LLM may instantiate real objects where mocks are needed or simply not inject anything at all, causing failing tests.

Interfaces, Abstractions, and Mocking Pitfalls

Interface-driven designs (a hallmark of clean architecture and SOLID principles) introduce extra indirection that LLMs struggle to fully resolve in tests. The model must infer the correct behavior of each interface implementation to decide what to assert or how to simulate it. Often, *LLM-generated tests only cover the "happy path"* where each interface returns a nominal value, and they fail to exercise alternative flows or exceptions. Empirical studies show that LLMs tend to misinterpret or oversimplify such abstractions – in one analysis of GPT-generated code, models commonly produced *incorrect logic or missed necessary conditions* in the code, indicating difficulty handling complex logic branchesprompthub.us. As a result, tests may neglect failure paths (e.g. when a repository interface throws an error or returns unexpected data) because the model doesn't fully reason through those multi-step interactions. Furthermore, misuse of mocking APIs is common: 57.9% of ChatGPT's auto-generated tests in a study could not even compile due to issues like calling non-existent members or wrong types (often stemming from incorrect use of dependencies or mocks)mingwei-liu.github.io. Such high failure rates underscore that rich OOP patterns with many interfaces present a consistent stumbling block.

Asynchronous Methods and Event-Driven Behavior

Applications in domains like fintech or insurance often rely on **async workflows and events** (e.g. processing transactions or issuing policy events) to handle operations. These pose a significant challenge for LLMs when generating unit tests. An illustrative example was observed in an Angular service test generation experiment (conceptually similar to C# event-handling): ChatGPT wrote a test that subscribed to an observable but placed the assertion *outside* the subscription callbackreadmedium.com. This meant the test could finish before the asynchronous event completed – a clear logical bug. The human had to correct the model by ensuring the expectation ran *inside* the event handler and using proper async test patterns (like a completion callback). This example highlights how **multi-step asynchronous logic easily confuses LLMs** – they may forget to await asynchronous methods, omit synchronization (Task.Wait or .Result in .NET), or not handle callbacks properly. In event-driven architectures (using C# events, delegates, or observer patterns), LLMs often fail to simulate the event triggering and listening mechanism. For instance, verifying that an event was raised might require attaching a test handler or using a mock observer, but a naive model-generated test might skip directly to an assertion without ever hooking into the event. These pitfalls align with research noting that LLM-generated code frequently *struggles with ordering and conditions* in complex flows, especially when timing is involvedarxiv.org. The result is tests that either falsely pass (missing the event logic entirely) or fail due to race conditions and unawaited tasks.

Multithreading and Concurrency Challenges

Concurrent and multithreaded components (e.g. background workers, parallel processing in a banking engine) add another layer of difficulty that LLMs do not handle well. Because LLMs lack true execution simulation, they tend to ignore thread-safety concerns or timing issues in generated tests. For example, if a method spins up multiple tasks or uses locks, a robust unit test might need to inject a controlled TaskScheduler or use synchronization primitives to test race conditions – nuances far beyond the straightforward code patterns LLMs have memorized. In general, current models have trouble reasoning about interleaved operations or the need for thread coordination. Prior analyses of code generation failures show that LLMs *struggle with corner cases and complex scenarios* that require reasoning about state over timearxiv.org. Deadlocks, data races, or event ordering issues are seldom anticipated in an AI-generated test. Thus, an ASP.NET Core service that uses multithreading (for example, a pricing engine updating caches in parallel) can "break" an LLM's logic – it will likely produce a simplistic test that doesn't account for concurrency issues (or it might avoid the issue entirely by not testing the concurrent aspect). This gap means such tests may miss bugs that only manifest under concurrent execution, which is a known blind spot for AI code generation.

Advanced Design Patterns (Mediators, Repositories, Strategies, Extensions)

Complex enterprise applications often employ patterns like the **Repository** (for data access), **Mediator** (for request/response dispatching via libraries like MediatR), **Observer** (event subscriptions), or **Strategy** (pluggable business algorithms). These patterns distribute logic across many components and require a test to coordinate multiple pieces – precisely the kind of multi-step reasoning that trips up LLMs. For example, if a service method under test uses a Mediator to send a command, a correct unit test should verify that the mediator was called with the right request and perhaps simulate a response. AI-generated tests, however, may *entirely ignore the mediator interaction*, asserting only the direct outputs of the method. This was noted in user trials: without explicit instruction, ChatGPT won't automatically assert that IMediator.Send() was invoked, because that involves understanding an indirect effect rather than a return value. Similarly, for the repository pattern, the model might call the real repository method (which is undesirable in a unit test) or assume a return value without setting up a mock to produce it. This leads to tests that either hit a database (violating unit test isolation) or fail due to null returns. **Strategy patterns** and polymorphic behaviors pose the challenge of covering multiple cases – an LLM might test only one concrete strategy and miss others. In general, **coverage of edge cases is weak**. All too often the AI sticks to a single representative scenario it saw in training data, missing alternative branches. Research confirms this tendency: LLMs often generate code that is *logically incomplete*, failing to account for less common conditions or error pathsarxiv.orgmedium.com. Even with extension methods – a C# language feature that might confuse the model – there are pitfalls. An LLM might treat an extension method as if it were a normal instance method (or vice versa), or forget the necessary using import for the static class, causing a compilation error (another contributor to that ~58% compile-error rate in generated tests)mingwei-liu.github.io. These intricate pattern-driven architectures thus supply fertile ground for LLM failure: the model's lack of true understanding leads it to oversimplify, resulting in tests that pass trivial scenarios but break or miss the mark for complex interactions.

Complex Domain Logic and Business Rules

When the domain itself is complex (banking calculations, insurance policy rules, etc.), writing good tests requires deep reasoning about the business requirements – something LLMs cannot reliably derive just from code. **LLMs often have no access to the specification or intent behind code**, so they guess expected behaviors from patterns in training data. This can lead to plausible-sounding but incorrect assertions. For instance, if a banking service method calculates fees, an AI might hard-code an expected fee value it "thinks" is right, but which doesn't actually match the business rule for certain inputs. Such failures have been noted in benchmarks: LLMs sometimes produce code (or tests) that is syntactically correct yet *semantically wrong*, because the model misunderstood the true requirementsmedium.com. Moreover, **edge cases in business logic** (like a boundary condition for insurance premium calculation) are frequently overlooked. A human tester would consider edge inputs (zero amounts, maximum limits, invalid data) and failure modes, but an LLM tends to focus on a generic typical case. One QA engineer observed that ChatGPT's suggested test cases can easily miss "silent mode" or background conditions in an app – analogous to missing subtle business scenarioskeploy.io. This aligns with academic findings that even advanced models like GPT-4 can *consistently fail certain reasoning tasks* if they haven't seen that exact pattern beforearxiv.org. In domains where correctness depends on multiple steps of reasoning or external knowledge, the model's limitations become evident: it cannot reliably infer all the necessary scenarios, leading to incomplete test coverage.

Evidence from Benchmarks and Community Feedback

Multiple evaluations and user experiences reinforce the above failure modes. A 2024 study systematically testing ChatGPT's unit test generation found that **only 24.8% of its generated tests ran successfully** without modificationmingwei-liu.github.io. The rest had issues – *57.9%*

had compilation errors (e.g. using wrong class/method names, type mismatches, or accessing privates), and another 17.3% compiled but then **failed at runtime** due to incorrect assertions or logic errorsmingwei-liu.github.io. These numbers confirm that non-trivial code often confuses the model. The same study noted that when tests *did* pass, they achieved coverage and readability on par with human-written tests, but reaching that point required addressing the reasoning gapsmingwei-liu.github.io. Community forums echo these findings: developers caution that blindly trusting AI-generated tests is dangerous, as the model may not truly understand the intended behavior. One practitioner flatly stated that generating unit tests from code alone is "generally a bad idea" because unit tests encode specifications and edge knowledge that an AI won't grasp automaticallyreddit.com. Across different models (GPT-4, Claude, Google's Gemini, etc.), the consensus is that **increasing model size or power doesn't fully eliminate these pitfalls**. All models share certain failure patterns, such as *missing conditional branches or producing logically flawed checks*, when confronted with complex codeprompthub.us. Some newer tools improve results by iteratively prompting the LLM or combining it with static analysismingwei-liu.github.io, but the fundamental challenges remain.

Crafting Adversarial Prompts Using these Insights

The architectural elements above provide a blueprint for stress-testing LLMs with **adversarial unit-testing prompts**. By constructing a hypothetical C# project that embodies these hard cases – e.g. a Clean Architecture **fintech service** with multiple injected interfaces, using a repository pattern and a Mediator, performing asynchronous operations that trigger events, and containing nuanced business rules with edge conditions – one can prompt an LLM to generate tests and observe the failure points. The known model weaknesses (not injecting all dependencies, not handling events or async timing, ignoring certain branches, using incorrect assertions) can be triggered by such a scenario. In essence, the more the code under test requires *multi-step reasoning* and understanding of interactions, the more likely a current LLM will stumble. By referencing public benchmarks and examples in prompt design (for instance, asking the model to achieve 100% mutation coverage on a methodstartearly.ai or to test a scenario known to trip up GPT-4's reasoning), researchers can systematically evaluate where the model's logic breaks. The goal of adversarial STEM prompts in this context is to **push LLMs to the edge of their reasoning capabilities**. Incorporating the patterns above into challenge prompts – and requiring the model to, say, correctly mock all collaborators with Moq, assert both success and exception cases, and handle an event callback – aligns with known difficult areas. As studies and community trials have shown, these complex structures consistently yield LLM mistakes, making them ideal for probing and improving the next generation of AI coding assistantsarxiv.orgmingwei-liu.github.io.