**Advanced Patterns to Confuse LLM Unit Test Generation**

**Overview:** Modern ASP.NET Core Web API projects can be intentionally designed with complex patterns that **confound Large Language Models (LLMs)** when they attempt to generate unit tests (e.g. using Moq and xUnit/NUnit). In addition to previously identified pitfalls, the following advanced software design constructs introduce **reasoning-related challenges** that often lead LLMs to produce incorrect or logically flawed tests.

**Complex Dependency Injection Chains**

**Deep or multi-level dependency injection (DI) hierarchies** make it hard for an LLM to understand the full object graph and interactions. When services depend on other services which in turn depend on further layers, the model can **"fail to understand the codebase structure and the dependencies between functions, classes, or modules"**arxiv.org, leading to misguided tests. For example, a controller that requires a service, which itself requires several repository and helper classes, creates a chain of dependencies that an LLM might not fully resolve. The model could **forget to mock a lower-level dependency or assume a wrong default**, causing compilation errors or tests that pass trivial paths only. Deep DI chains also often rely on the IoC container for instantiation, which an LLM may not set up correctly in a unit test context.

*Why it breaks LLM reasoning:* The model must infer how to construct the class under test through multiple layers of abstraction. Miss one piece, and the test either fails to compile or doesn't actually exercise the target logic. In practice, LLMs tend to prioritize high-level logical structure over boilerplate setup and thus may omit necessary dependency configurationarxiv.org. This results in tests that don't properly instantiate the subject under test or that stub out important behaviors, ultimately yielding meaningless or false-positive tests.

**Polymorphic Interfaces and Multiple Implementations**

Using **interfaces with multiple concrete implementations** (e.g. strategy or factory patterns) can reliably confuse an LLM's test generation. In the strategy pattern, for instance, you define an interface with several implementations and choose between them at runtime based on some conditionlinkedin.com. This means the correct behavior of the system depends on which implementation is in use. LLMs often struggle to reason about **polymorphic behavior** – they might write a test using a different implementation than the one actually used in the scenario, or not realize that a specific input triggers a specific implementation.

For example, suppose an interface INotificationSender has both EmailSender and SmsSender implementations. The code might select one via configuration or input. An LLM-generated test could ignorantly **inject the "wrong" implementation** or try to test both in cases where only one is relevant, leading to irrelevant or failing tests. The strategy/factory approach "allows selecting different implementations based on conditions or parameters"linkedin.com, which means a unit

test must ensure the right one is chosen for a given scenario. An LLM without precise reasoning might not set up those conditions correctly, producing tests that either never execute the target implementation or that make redundant assertions.

*Why it breaks LLM reasoning:* The model has to track conditional logic that swaps out implementations. If this selection logic isn't straightforward, the LLM might misunderstand which code path is active. It may also **hallucinate behavior** from an implementation that isn't actually used. This results in tests that either do nothing useful (e.g. always pass because the wrong branch was tested) or tests that fail because the LLM expected a different implementation's behavior. Ensuring the correct interface implementation is exercised requires understanding config or factory logic that an LLM often fails to deduce on its own.

**Asynchronous and Concurrent Workflows**

**Async methods, multithreading, and concurrency** introduce timing and ordering complexities that are difficult for LLMs to handle in unit tests. If the code under test uses async/await heavily or spawns parallel tasks, a naive LLM test might forget to await tasks or not properly handle asynchronous completion. In some cases, LLMs produce tests that call an async method but don't await it, leading to tests that erroneously always succeed (the test finishes before the async work throws an error) or flakiness. Properly testing async code often requires using the correct test framework support (like Assert.ThrowsAsync or using an async test method), which an LLM might not utilize correctly, given the nuances in xUnit vs NUnit support for async methodslearn.microsoft.com.

**Concurrency pitfalls** (race conditions, thread synchronization issues) are even trickier. By nature, race conditions are nondeterministic and "there's no good general way to trigger a race condition in testing. Your only hope is to design them completely out of your system."softwareengineering.stackexchange.com. LLMs, lacking true understanding, can't devise reliable strategies to test thread safety or race conditions. They might either ignore the concurrent aspect entirely or attempt a simplistic approach (like running two threads without proper coordination), which could result in brittle or ineffective tests. For example, code that uses locks or shared state might require careful ordering in tests to simulate a race – something far outside an LLM's typical capabilities.

*Why it breaks LLM reasoning:* Handling asynchronous behavior requires the model to reason about **time and order**, which is not its strength. Concurrency issues involve subtle interleavings that even experienced developers struggle to test. An LLM might not foresee the need to e.g. wait for background tasks to complete, leading to tests that pass erroneously. Or it may not understand the necessity of injecting a test scheduler or synchronization context. These oversights yield tests that either deadlock, never catch the bug, or produce false failures. In

summary, async and multithreaded workflows present a minefield of timing-related logic that often defeats the pattern-matching nature of LLMs.

**Event-Driven and Indirect Behaviors**

**Event-driven architectures** (using C# events, callbacks, or observer patterns) and other **indirect execution** mechanisms (such as mediator/pub-sub patterns or extensive use of extension methods) create non-linear code flows. Instead of method A directly calling method B, the code might raise an event or send a message that eventually triggers B. This decoupling confuses LLMs, which prefer more explicit call graphs.

Traditional .NET events, while less common in modern code, are "still used by many libraries (legacy and not)"[docs.educationsmediagroup.com](), and testing them involves subscribing to events and possibly using Moq's event features. Moq **can** support raising events on mocks following the standard patterns (like event handlers based on EventArgs)[docs.educationsmediagroup.com](), but the LLM has to know how to use mock.Raise(...) or the proper event invocation – a detail it often misses. As a result, an LLM might write a test for event-driven code that **never attaches a handler or never triggers the event**, thereby not actually testing the outcome of the event. Event-driven code may also execute asynchronously on a thread pool (for example, using Task.Run inside an event handler), compounding the difficulty.

**Observable/reactive patterns** (using IObservable<T> or reactive streams) are similarly problematic. The test needs to subscribe to the observable and wait for emissions, possibly using a testing scheduler. LLMs not specifically trained in Rx.NET nuances might ignore necessary steps, leading to tests that either hang or miss the event.

**Mediator or pub-sub patterns** (e.g. using MediatR library or a custom mediator) hide direct method calls behind a message dispatch. For instance, a controller might Send() a command via MediatR instead of directly instantiating a handler. The unit test then must either mock the mediator or test the handler in isolation. An LLM might not recognize this separation and could attempt to call the handler method directly (bypassing the mediator), or vice versa, not realizing it needs to ensure the mediator dispatches to the correct handler. This results in tests that don't align with the actual code execution path.

**Extension methods** present another form of indirection. An extension method is essentially a static method disguised as an instance method. These are **not easily mockable** (since "extension methods are static, and statics aren't mockable"[softwareengineering.stackexchange.com]()). If critical logic is implemented as an extension (for example, an extension that manipulates an HttpContext or a domain object), an LLM-generated test might struggle: it cannot use Moq to intercept the static call. The model might ignore the extension's effect entirely or attempt to

fake it in convoluted ways. In general, any **static or external call** that doesn't have an interface abstraction will be a blind spot for an LLM, often resulting in missing verification of those side effects in its tests.

*Why it breaks LLM reasoning:* Indirect patterns require the model to track cause-and-effect through decoupled components. The LLM has to infer that "event X causes outcome Y" even though Y is executed in a different place or time. With mediator or events, the triggering and handling are separated, which LLMs might not connect properly. They may write tests that assert nothing or test the wrong unit (e.g. test only the mediator stub rather than the real handler logic). The extension methods add to this by hiding actual calls – an LLM might treat an extension call as a black box and not account for its behavior in the test. All these factors yield tests that either miss critical assertions or simply don't execute the important code at all.

**Conditional Middleware and Pipeline Branching**

ASP.NET Core's middleware pipeline can have **conditional branches** based on configuration or the incoming request. For example, you might use app.UseWhen(...) or environment checks to add certain middleware only in specific circumstances (like enabling a developer exception page in Development, and a custom error handler in Production). Middleware can also short-circuit the pipeline – e.g., an authentication middleware that returns a 401 response and **does not call the next middleware** if a request is unauthorizeddevtrends.co.uk.

Such dynamic behavior is hard for an LLM to reason about in unit tests. The correct test might need to simulate different environments or request paths to cover each branch. A naive LLM might generate only a generic test for the middleware's happy path and completely miss the conditional branch logic. Or it might attempt to test middleware in isolation without setting up the necessary HttpContext or pipeline, leading to null references or no effect.

**Example:** Consider a middleware that checks for an API key header: if the header is missing or invalid, it writes an error response and does not call _next(). Otherwise, it calls _next() to continue the pipeline. An LLM could easily overlook testing the branch where the API key is missing, or fail to assert that the next middleware was or wasn't invoked. It might also not realize that to test middleware, one should construct a dummy HttpContext or use an HttpMessageInvoker/TestServer – not trivial steps.

*Why it breaks LLM reasoning:* The model must understand global configuration and request-specific logic, which is context often outside the code snippet of the middleware itself. LLMs tend to generate tests that are **too superficial**, for instance checking only that some outcome occurs for one scenario. Conditional logic requires multiple test cases with different setups. Additionally, because middleware often involves static setup (the Startup.Configure pipeline) and extension methods to register, an LLM might not set up the test harness correctly. This can

produce tests that compile but always pass (not actually invoking the middleware logic) or tests that fail due to an improperly simulated environment. In short, pipeline branching introduces **global state/context** that is hard to encapsulate in a prompt, thus tripping up the test generation.

**Hidden or Unobservable Side Effects (Logging, etc.)**

Code that relies on **side effects rather than return values** is a known challenge for unit testing – and by extension, for LLMs writing those tests. A common example is heavy use of logging. Methods that perform important work might only **log warnings or errors** without returning distinct error codes. Testing such methods requires asserting that the logger was called with the expected messages. This means the test must capture or mock the logging behavior. LLMs often fail to correctly verify log output because it's not a direct functional outcome.

For instance, a service method might catch an exception and log "Error XYZ occurred" then return null. An LLM-generated test might only check that null is returned, **ignoring the log**, even though the log is the primary indicator of the handled error. Or if it does attempt to verify logging, it needs to use Mock<ILogger>.Verify(...) with the correct parameters – a non-trivial setup that requires understanding of ILogger extension methods and It.Is<It.IsAnyType>((o,t) => ...) matching. Setting up these verifications *"demands careful effort and attention" and gets complex if multiple log statements are involved*medium.com. LLMs frequently get the Moq syntax wrong for verifying ILogger calls, or assert the wrong message (due to slight hallucination in log text). This leads to tests that either don't actually catch whether logging happened, or that are brittle (failing because the log message text mismatches exactly).

Other unobservable side effects include writing to an in-memory cache, performance counters, or triggering an external call without a return value. In all such cases, the **effect isn't directly visible** from the method's output. A human tester might solve this by injecting an interface (e.g. an ICache or using a fake logger) and then asserting on that, but an LLM might overlook that entirely. For example, if a function writes to a static cache, an LLM might not consider verifying the cache state after the call (or might not know how to access it). This means the generated test does nothing meaningful – it might only call the function and then assert that the result is not null (which says nothing about the cache being updated).

*Why it breaks LLM reasoning:* LLMs are primarily trained on **patterns of input-output** rather than truly tracking invisible state changes. If the effect of a function isn't apparent from the return value or a thrown exception, the model may not realize a test should verify something else. Logging is a prime example where the "output" is to an external sink. The complexity of capturing and asserting log messages (without causing too brittle tests) is something even human developers handle carefully (often using in-memory log providers or custom test

sinks)medium.com. An LLM generally won't invent a whole fake logger provider unless explicitly told. Thus, side-effect-heavy code often yields LLM tests that miss the core verification (making them logically flawed), or the LLM attempts an assertion but implements it incorrectly.

**Assessing the Connected Repository for These Patterns**

The provided GitHub repository can be analyzed to see which of the above patterns it already contains, and how it might be extended into a **"model-breaking" prompt** for unit test generation. From inspection, the repository (an ASP.NET Core Web API project) already exhibits some complexity common to such projects:

- **Dependency Injection:** The project uses ASP.NET Core's DI container to inject services into controllers (e.g., via constructor injection). This means the basic DI pattern is present. If the current DI setup is relatively shallow (e.g. controller -> service -> repository), it could be **made deeper** by introducing additional layers (wrapping services inside other services, or adding decorators). This would create the *complex DI chain* scenario described earlier. Since LLMs already struggle with understanding dependency graphs, using the repo's DI setup in a prompt (especially if extended with more layers or conditional resolution) could induce test generation failures.
- **Multiple Implementations:** We should check if any interface in the repo has multiple registered implementations. For example, there might be an interface for data access with both a production implementation and a development stub. If not already present, this is straightforward to add – one could introduce an interface (e.g. INotifier) with, say, an EmailNotifier and SmsNotifier, and use a factory or config to choose one at runtime. Adapting the repo to include such a pattern would set a trap for LLMs, as discussed above, because the model may not test the correct implementation. If the repo already uses a strategy pattern or similar (for instance, different implementations for an interface based on entity type or business logic), that part can be directly used in a prompt to challenge the LLM.
- **Async Workflow:** The repository likely uses async/await (as is idiomatic in ASP.NET Core, e.g. for database or HTTP calls). These asynchronous methods can be directly used in a prompt to observe if the LLM knows to await them in tests. If the repo's code includes any fire-and-forget tasks or parallel processing (for example, launching background tasks or using Task.WhenAll), that's even better to showcase concurrency challenges. If not, adding a few asynchronous methods or parallel loops in the business logic can simulate the *multithreading pitfalls*. The adapted prompt could then check if the LLM tries to handle those (likely it won't correctly, given the difficulty of testing concurrent behavior).

- **Event-Driven Components:** Many Web API projects might not use C# events or observables by default, but if this repo uses a library like MediatR for CQRS, that is essentially an event-driven (mediator) pattern. We should see if the codebase has any domain events or uses INotification/INotificationHandler or similar constructs. If yes, those are prime candidates for confusing the LLM – the prompt could include a snippet where an action triggers an event and another component handles it. If the repo doesn't currently have an event-driven mechanism, it could be extended (for example, define a custom event on a service or use an Action<T> callback) to simulate this scenario. The presence of any **indirect call patterns** (even something like an IHttpClientFactory where the actual HTTP call is done outside the service) can contribute to this category. The repository can be adapted to include a few such indirect behaviors if needed (e.g., an extension method that performs some operation used across the code) to see if the LLM picks up on those details.
- **Middleware and Conditional Logic:** The startup (Program.cs or Startup.cs) of the project likely has environment-based branches (e.g., if (env.IsDevelopment()) to use the Developer Exception Page). That is already a conditional branch in the request pipeline. Including that in a prompt might cause an LLM to produce a single test that doesn't account for environment differences. Additionally, if the repo has any custom middleware components, those could be included. We should check if there's any usage of app.UseWhen(...) or similar. If not, adding a simple conditional middleware (for example, only enable a certain header processing for specific path or tenant) in the code can be done to amplify this effect. The goal is to ensure the prompt contains at least one piece of middleware logic that the LLM would have to simulate in a unit test (which is non-trivial). The repository's existing configuration can likely be directly used, since even the out-of-the-box template has some conditional middleware – making it a candidate for a model-breaking prompt portion.
- **Logging and Side Effects:** The repository almost certainly uses ILogger<T> in controllers or services to log errors or information. This is an unobservable side effect that, if included in a test prompt, will check whether the LLM tries to verify logging. We expect it might ignore the logs. The repo's code where logging is done (e.g., logger.LogWarning("X happened")) can be part of the prompt. If the LLM does attempt to test it, it might get the Moq verify syntax wrong, which would be a clear failure. Beyond logging, if the repo uses any caching (e.g., in-memory cache) or similar, those can be highlighted. The existing code can likely be used directly here – many projects log and then return a result. That should suffice as a subtle trap for the LLM's reasoning. If needed, we can tweak the code to log multiple messages in one method (increasing the

complexity for verifying order or content of logs, echoing the point that verifying "multiple log statements" is complex without proper toolsmedium.com).

**Conclusion:** The connected repository already contains several of these hard-to-test patterns (at least DI and async, logging, etc.), and can be extended with others (like multiple implementations or explicit event-driven code) to create a truly complex prompt. By incorporating these elements into one ASP.NET Core project, we increase the likelihood of an LLM-generated unit test suite failing. The **combination of deep dependency graphs, polymorphic behavior, async concurrency, indirect event handling, conditional execution, and side-effect-laden methods** presents a worst-case scenario for LLM reasoning. This can directly be used as a "model-breaking" prompt: when the LLM is asked to produce unit tests for such a codebase, it will likely produce incorrect or incomplete tests, validating the effectiveness of these patterns in challenging AI-based code understanding.

Overall, **architectural complexity and indirection** are key: the more the code relies on understanding context, hidden flows, or multi-step reasoning, the more likely an LLM will stumble. By leveraging and extending the patterns identified above, the repository can serve as a robust testbed to systematically **cause LLM-generated unit tests to fail** in logically substantive ways, not just minor syntax errors. Each pattern exploits a known weak point in LLM reasoning – from failing to follow dependency chains to mis-handling asynchronous operations – thereby providing insight into the boundaries of current AI coding assistance.

**Sources:**

- LLM limitations in understanding project structure and dependenciesarxiv.org
- Strategy pattern with multiple implementations (runtime-selected)linkedin.com
- Extension methods as static calls (not directly mockable)softwareengineering.stackexchange.com
- Events in .NET and Moq's support (indirect usage)docs.educationsmediagroup.com
- Conditional middleware pipeline example (branching/short-circuiting)devtrends.co.uk
- Challenges in verifying logging with mocks (side effects)medium.com
- On the difficulty of testing race conditions (concurrency)softwareengineering.stackexchange.com