Suitability of the Repository for a Complex ASP.NET Core API Unit Test Challenge

Dependency Injection and Service Layers

The repository's design makes heavy use of dependency injection with multiple service layers. For example, the core logic class FooLogicImplementer expects an IEnumerable<IFooCollection> in its constructor, meaning it can depend on **multiple** implementations of IFooCollection at oncegithub.com. In the provided Azure Functions Startup, two different IFooCollection implementations (FooLogic1 and FooLogic2) are registered in the DI containergithub.com. This kind of setup (multiple injected services of the same interface) is known to confuse LLMs. When generating tests, models often **omit required dependencies or instantiate the class incorrectly**, e.g. forgetting to provide one of the constructor parameters or assuming a default constructor exists. They might even try to use a DI container in the unit test (which is not how unit tests supply dependencies). The repository's pattern – a logic class depending on a *collection* of interfaces (each encapsulating further dependencies like IFooWork) – mirrors the "multiple service layers" scenario noted in research. This **cascading dependency graph** requires setting up numerous mocks and wiring them in a list, a setup that LLMs often mishandle. In short, the repository already contains the kind of DI complexity (many injected services) that reliably trips up models in unit test generation.

Interfaces and Polymorphism

The code heavily uses interfaces and polymorphic behavior. There are abstractions for collections and work units (IFooCollection, IFooWork, plus the logic interface IFooLogicImplementer), with multiple concrete classes implementing them (two IFooCollection implementations, each using a different IFooWork). This design requires context-aware mocking – the test must decide which concrete implementations or mocks to provide. LLMs frequently become confused about which layer to mock or which concrete type to use for a given interface. For instance, a common mistake is **"mocking the wrong thing."** If one class calls into another interface, the AI might try to mock the higher-level interface instead of the lower-level dependencyfile-v3ynhhdenxb5t61zexbjry. In our case, a naive AI might attempt to mock FooLogicImplementer itself or a FooCollection in the wrong way, rather than providing mocks for the underlying IFooWork instances. The repository's polymorphism (multiple implementations of one interface and nested interface calls) means a test must provide the correct context – e.g. ensure each IFooCollection returns registrations each containing an IFooWork. **LLMs often lack this context**, sometimes using the wrong concrete classes or forgetting to supply any implementation at all. The presence of these interface-driven patterns in the repo is a strong source of confusion for automated reasoning, which is exactly what we want (the model might, for example, not realize it needs to **mock each IFooWork inside each IFooCollection**, leading to NullReferenceExceptions or untested branches).

Asynchronous Methods and Task Return Types

Asynchronous logic is present throughout the code. The FooLogicImplementer.ExecuteLogic() method is async and awaits each registration's Exec() (which in turn awaits an IFooWork.CheckFooAsync call)github.com. The unit tests in the repository handle this by using Task.FromResult in their Moq setups – for example, returning a completed FooResult from CheckFooAsyncgithub.com – and by awaiting the result of ExecuteLogic()github.com. These details are precisely the nuances that LLMs often get wrong. It's documented that AI-generated tests frequently **forget to await async calls or use incorrect Moq setups for tasks**, causing compilation errors or tests that pass trivially without exercising the async codefile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry. In fact, one common error is using .Returns(expectedFoo) instead of .ReturnsAsync(expectedFoo) when mocking an async methodfile-v3ynhhdenxb5t61zexbjry. The repository's tests explicitly use Task.FromResult to wrap resultsgithub.com, which is a subtle detail an LLM might omit. Given this design, an LLM must properly await the ExecuteLogic call and use ReturnsAsync (or an equivalent) for CheckFooAsync. Past experience shows models often **miss these async patterns**, e.g. calling the async method without awaiting it or not wrapping the result in a Task, leading to incomplete test logicfile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry. Therefore, the asynchronous workflow in this repo (multiple awaited calls in a loop) is likely to trip up the model's reasoning without explicit guidance.

Factory/Strategy Patterns in DI Context

The architecture exhibits a strategy-like pattern: multiple IFooCollection implementations are registered and used, effectively providing different "strategies" or sources of FooRegistration data. While the code doesn't explicitly use a factory method or strategy selector at runtime (the collections are all used in aggregate), it sets the stage for such patterns. In a similar vein, another part of the repository or a related sample uses an IFooCalculatorStrategy chosen by a **provider** parameter (as seen in a FooController example, which selects a strategy based on a string like "base"). This aligns with the Strategy pattern where different algorithms are plugged in based on input. LLMs struggle with such patterns because writing a test requires understanding which implementation will be picked and how to configure the DI container or factory accordingly. Our current repo already forces the model to reason about multiple implementations in the DI container (all IFooCollection instances get injected). For an even more confusing scenario, we could introduce a **factory** that selects a specific IFooCollection or IFooWork at runtime based on some input. However, even without an explicit factory method, the model must grasp that *two* implementations of IFooCollection exist and both contribute to the logic. This distributed logic (a form of manual strategy combination) is exactly the kind of multi-step reasoning that often trips up LLMsfile-v3ynhhdenxb5t61zexbjry. In summary, the

repository is already largely suitable here – it uses multiple strategy-like components via DI – but if needed it can be minimally adapted (for example, by adding a simple factory or strategy selector method) to further ensure the model faces a non-trivial decision about which implementation to use.

Complex Controller Input/Output Behavior

One area that could be **augmented** is the presence of a complex Web API controller or endpoint logic. Currently, the core logic is encapsulated in services and an Azure Functions startup, but we don't see an actual ASP.NET Core **Controller** class in this repo snippet. To fully emulate a "complex C# web API", we might wrap the FooLogicImplementer in a controller action that handles HTTP input and output. For instance, consider a controller method that accepts a request model and a query parameter (like the FooController example with a provider parameter and model binding for FooRequest). It could validate the model (ModelState.IsValid check), choose a strategy or set a default provider, invoke the IFooLogicImplementer, then map the result to an HTTP response. Such a controller would introduce conditional branches (valid vs. invalid model, different provider strings) and possibly model-to-domain mapping. **LLMs often falter on these front-end aspects**. They might not realize they need to simulate model validation failing, or they assume default inputs that bypass certain branchesfile-v3ynhhdenxb5t61zexbjry. For example, if there's a branch that returns BadRequest when the model is invalid, an AI might never test that path (always assuming valid input) – analogous to how they often assume default configurations and miss error branchesfile-v3ynhhdenxb5t61zexbjry. The absence of an HttpContext or proper request setup is another common pitfall in testing controllers/middlewarefile-v3ynhhdenxb5t61zexbjry. If we add a controller, the model would need to construct a request (or relevant context) – something GPT models frequently omit, leading to null reference errors in testsfile-v3ynhhdenxb5t61zexbjry. **Minimal additions** could make the repository even stronger here: for example, creating a simple FooController that calls the IFooLogicImplementer and has a couple of branches (one branch if all results are good vs. another if any are bad, or a query param that affects logic). This would force the LLM to deal with HTTP input, possibly [FromQuery] or [FromBody] parameters, and model state – all of which increase the likelihood of mistakes in the generated tests (like not providing necessary context or not asserting the correct HTTP status).

Event-Driven Logic and Domain Events

The current repository doesn't explicitly implement domain events or observer patterns, but adding a touch of **event-driven behavior** could further confuse the model with minimal effort. For instance, FooLogicImplementer could publish an event or call an INotifier/IEventPublisher service when a certain condition is met (say, when any FooResult is bad). This would introduce an **observable side-effect** that a well-written unit test should verify (e.g. ensuring that the event

publisher was invoked), but which an AI might ignore. Studies have noted that when code uses patterns like the Observer (event subscriptions) or Mediator, LLM-generated tests often fail to assert the indirect effectsfile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry. They tend to focus only on return values and forget to verify that an event was fired or a mediator call was madefile-v3ynhhdenxb5t61zexbjry. By adding a domain event or callback, we'd be exploiting this weakness. The model might either omit setting up a mock for the event publisher (leading to a null call) or simply not assert that the event occurred – a **common oversight** since it requires understanding an indirect outcome instead of a direct returnfile-v3ynhhdenxb5t61zexbjry. Minimal change needed: introduce a new interface (e.g. IFooEventPublisher) and inject it into FooLogicImplementer. Inside ExecuteLogic(), call something like eventPub.Publish(result) in the branch where FinalRes is "Bad". This small addition would engage the Observer pattern. It aligns with the known tricky patterns (the PDF explicitly cites Observer and Mediator patterns as tripping up LLMs)file-v3ynhhdenxb5t61zexbjry. Overall, while the repo is already complex, **this tweak would ensure even more reliably that an AI test generator misses something important** (either by not providing or not verifying the event interaction).

Mockable Persistence or External Service Abstractions

The design already includes an abstraction (IFooWork) that stands in for an external operation – it could be imagined as a remote service call or a database check. In the sample tests, IFooWork is mocked with specific return values (good or bad results)github.com. This is essentially a **persistence/external service abstraction** being mocked, akin to a repository or API client. LLMs often stumble here by either not realizing they need to mock such dependencies or by trying to call the real implementation in the testfile-v3ynhhdenxb5t61zexbjry. In fact, failing to mock external services (or mocking the wrong level of abstraction) is a top cause of AI-generated test failures. The repository could be extended to include a more explicit **Repository pattern** – for example, an IFooRepository that FooWork might call to retrieve some data, or an external HTTP client interface. However, this might be overkill, since the effect is similar: we already have a multi-layer call (Logic -> Registration -> Work -> returns result). If we did want to incorporate it, a minimal addition could be to have FooWork accept an IDatabase or IExternalService interface, and use that inside CheckFooAsync. This would force the LLM to provide a mock for that lower-level dependency as well. As noted, when faced with a Repository pattern, models might *call the real repository method or assume a return value without setting up a mock*file-v3ynhhdenxb5t61zexbjry – exactly the kind of failure we want. In summary, the repo's existing abstractions are already sufficient to pose a challenge (the LLM must mock IFooWork correctly for each registration, and not forget any). If needed, adding one more layer (a repository or external API interface) would be a **minimal change** to increase complexity, ensuring the model has yet another dependency to account for. Given that forgetting or mishandling such

dependencies is a known failure mode,file-v3ynhhdenxb5t61zexbjry, this would further guarantee a “model-breaking” unit test scenario.

Conclusion and Recommendations

Overall, the connected repository is **well-suited** as a foundation for constructing a complex C# ASP.NET Core API that breaks LLM unit test generators. It already implements multiple tricky patterns: heavy DI with many injected services, interface-driven architecture with polymorphic behavior, asynchronous workflows, and conditional logic in the service layer. These align with known pain points where LLMs often produce incorrect or incomplete tests (e.g. missing mocks, using wrong setups, not covering certain branches)file-v3ynhhdenxb5t61zexbjry. To fully realize the “model-breaking” potential, a few **minimal additions** could be made without drastically changing the design:

- **Introduce a Web API Controller** that uses the service layer, to incorporate model binding and branching logic (e.g. input validation or different execution paths). This will test the model’s ability to handle HTTP context and input conditions (areas where AIs often assume defaults or forget setup)file-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry.
- **Add a Domain Event or Mediator call** in the service logic (such as publishing an event on certain outcomes). This small addition forces the model to consider indirect side effects; otherwise the AI may ignore them, as has been observed with unverified mediator or event calls in AI-generated testsfile-v3ynhhdenxb5t61zexbjry.
- **Optionally, add one more abstraction** like a repository or config interface to the IFooWork or logic layer. This would reinforce the need for deep mocking. Given that AIs often fail to mock all necessary layers (or mock the wrong layer)file-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry, it will increase the likelihood of a failure in the generated test.

Crucially, none of the above requires overhauling the architecture – they are incremental enhancements in line with the existing design. The current code already embodies the complexity (multiple layers and interactions) needed to confuse an LLM. With the suggested tweaks, we ensure that **virtually every tricky aspect** – DI, polymorphism, async, strategy selection, controller logic, events, and external services – is represented. Such a combination is very likely to overwhelm a model like *o3-mini* and cause it to produce flawed unit tests, either by **omission, incorrect assumptions, or logical errors** in the test code. In conclusion, the repository is suitable and can be adapted with minimal changes to create a robust “LLM-breaking” unit test prompt scenario, leveraging exactly the patterns that are known to mislead current large language models in reasoning about C# unit testsfile-v3ynhhdenxb5t61zexbjry.

Explore the 'test' Repository for Complex C# Unit Testing Patterns

Complex C# Code Patterns in the 'test' Repository and Unit Testing Challenges

The **'test'** repository contains several C# projects that exhibit sophisticated patterns – from heavy use of dependency injection to strategy-based designs. These codebases align with **ASP.NET Core-style** architecture and include multiple abstractions, asynchronous methods, and extensive use of interfaces. Below we identify two major components and discuss their testing-relevant features, along with why they can **confuse LLMs** (Large Language Models) when generating unit tests.

1. FooLogicImplementer – Multi-Layer DI with Async Logic

Description: This component (found in the MoqXunitWithDI project) is an **Azure Functions/ASP.NET Core-style** service that heavily uses **Dependency Injection (DI)**. The FooLogicImplementer class depends on an **enumerable of interface implementations** – it takes an IEnumerable<IFooCollection> in its constructorfile-v3ynhhdenxb5t61zexbjry. In the function startup, multiple services are bound to the same interface (two implementations of IFooCollection are registered)github.com. Internally, FooLogicImplementer gathers data from all injected sources and processes them asynchronously. For example, its ExecuteLogic() method iterates through a list of FooRegistration objects and **awaits** each one's Exec() operationgithub.com. This yields a final result indicating if all outcomes were "Good" or if any were "Bad". The design exemplifies a **clean architecture** approach: the logic is separated from instantiation concerns, and abstract interfaces (IFooCollection, IFooWork, IFooLogicImplementer) define the behavior.

Testing-Relevant Components: Testing FooLogicImplementer requires assembling a complex object graph. Because it depends on **multiple services and nested collaborators**, a unit test must provide **multiple mock implementations**. In practice, each IFooCollection in the input list needs to be a mock that returns a set of FooRegistration objects (and each FooRegistration contains an IFooWork to be mocked as well). The repository's unit tests demonstrate this setup using **Moq**: they create a mock IFooCollection, then prepare two mock IFooWork instances and inject them via FooRegistration objectsgithub.comgithub.com. The mocks are configured to return pre-defined FooResult values (using Task.FromResult to wrap the result in a completed Task) so that ExecuteLogic() can run through its async loop deterministically.

Why LLMs Struggle: This pattern combines **multiple indirections and async calls**, which is challenging for an AI to reason about. The model must infer that it should create *multiple* mocks and assemble them into a list before calling the constructor. In practice, human testers will do exactly that – e.g. create several IFooCollection mocks each returning appropriate registrations – but LLMs often **fail to reason through this setup**file-v3ynhhdenxb5t61zexbjry. Common

mistakes include forgetting some dependencies, attempting to call a nonexistent default constructor, or not actually injecting the mocks (creating them but never using them)file-v3ynhhdenxb5t61zexbjry. Indeed, the example in the repository mirrors these pitfalls: an AI might omit adding one of the IFooCollection objects to the list or neglect to provide the inner IFooWork objects, leading to null references or incomplete testsfile-v3ynhhdenxb5t61zexbjry. Another challenge is handling the **asynchronous** aspect correctly. The repository's tests use Task.FromResult(...) and await the ExecuteLogic() call, but an LLM might **mis-handle async** – for instance, by setting up a mock to return a raw value instead of a Task, or by calling the async method without awaiting itfile-v3ynhhdenxb5t61zexbjry. Such mistakes result in either compilation errors or tests that erroneously pass because the async work never actually executes. In short, the **heavy DI + async** pattern here is a minefield for automated reasoning. As noted in the documentation, LLM-generated tests often miss dependencies or use mocks superficially in these casesfile-v3ynhhdenxb5t61zexbjry, making this component an excellent test of an AI's unit testing skill.

2. FooController & Strategy Pattern – Layered Web API with Dynamic Providers

Description: Another codebase in the repo (the Ttf.WebApi project) is a **Web API** application that follows a layered, somewhat **clean architecture** approach. It uses an external DI container (StructureMap) to wire up controllers and business logic. The FooController is an ASP.NET Web API controller that depends on two services: an IMapper (from AutoMapper) and an IFooCalculatorStrategygithub.com. The IFooCalculatorStrategy is part of the **business layer** and implements a **Strategy pattern**. Its job is to select the appropriate IFooCalculator implementation based on a provider key. Multiple concrete strategies (IFooCalculator implementations) exist, and they are all registered with the DI container. The strategy class (FooCalculatorStrategy) is constructed with an **array of IFooCalculator** instances injectedgithub.com. When the controller calls fooCalculatorStrategy.Calculate(...), the strategy in turn finds the calculator whose AppliesTo() method matches the given provider and delegates the calculation to itgithub.com. This design allows new strategies to be added without changing the controller or the strategy class (open/closed principle)github.com. The StructureMap configuration scans the business layer assembly and automatically registers all implementations of IFooCalculator and the strategy itselfgithub.com.

Testing-Relevant Components: The FooController and its strategy illustrate a use of **abstract factory/strategy patterns** in combination with DI. To unit test the controller in isolation, one would **mock the IMapper and IFooCalculatorStrategy**. The repository includes an example NUnit test (TestFooController) that does exactly this: it creates mocks for the mapper and the strategy, injects them into a new FooController instance, and then calls the Get methodgithub.com. The test verifies that the controller uses these dependencies – i.e. it checks

that the mapper's mapping method and the strategy's Calculate method were invoked with the expected parametersgithub.com. This kind of test focuses on the **interaction**: ensuring that the controller is delegating work to the strategy (and not, say, trying to compute things on its own). If we were to test the strategy class itself, we would provide it with a set of fake IFooCalculator implementations (or mocks of IFooCalculator) to see that it picks the correct one based on a provider string.

Why LLMs Struggle: This scenario introduces **indirection via design patterns**. The controller doesn't know about concrete calculators – it relies on a strategy interface, which in turn relies on multiple implementations of another interface. Such indirection can easily **confuse an LLM's reasoning**file-v3ynhhdenxb5t61zexbjry. When asked to write a test, a less sophisticated model might "get lost" and attempt to instantiate or call the wrong things. For example, instead of stubbing the strategy, an LLM might try to directly instantiate a specific FooCalculator class (bypassing the strategy selection logic), thereby not actually testing the intended patternfile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry. Another common failure is **mocking the wrong component**. In this case, the correct approach is to mock IFooCalculatorStrategy (or provide a fake that uses real IFooCalculator if we want to test the strategy logic itself). An AI, however, might mistakenly decide to mock the lower-level IFooCalculator implementations while still calling the real strategy, or vice-versafile-v3ynhhdenxb5t61zexbjry. This can result in a test that doesn't isolate the unit correctly – e.g. leaving the factory/strategy behavior untested or invoking real logic inadvertently. The repository's design also uses an external DI container and mapper, which add another layer of complexity: LLMs often are unsure how to handle these and might ignore the DI setup entirely or try to manually new-up classes that should be injected. The presence of the **Strategy pattern** (and an Abstract Factory-like registration of calculators) is specifically highlighted as a pitfall for AI-generated testsfile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry. The model needs to trace through the indirection – understanding that to test the controller's logic, it must simulate what the strategy would do. This is non-trivial without a deeper understanding of the code's intent. Overall, the FooController + strategy module provides **material to challenge an LLM's unit test generation** by requiring it to handle multiple abstraction layers correctly and not "short-circuit" the designed pattern.

Conclusion: LLM Unit Test Challenges in this Repo

Both of the above codebases are **ripe for exposing the weaknesses of AI in unit test writing**. They feature complex object interactions that an LLM must set up: multi-mock dependency injection, **async workflows**, and runtime selection through patterns like Strategy. The repository's content can be used to probe whether an LLM understands how to **provide all necessary dependencies and mocks** for a class under testfile-v3ynhhdenxb5t61zexbjryfile-

v3ynhhdenxb5t61zexbjry, and whether it can follow the intended control flow through factory/strategy indirections without being misledfile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry. In summary, the testing-relevant components here (like FooLogicImplementer and FooController/FooCalculatorStrategy) embody the kinds of **advanced C# patterns** – heavy DI, multiple interfaces, asynchronous calls, and design-pattern-driven architecture – that frequently **trip up automated reasoning** in unit test generation. This makes the 'test' repository an excellent source of challenging scenarios for Large Language Models, as evidenced by common mistakes documented (e.g. missing awaits, incorrect mock setups, or bypassing factory logic) when models attempt to write tests for such codefile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry.

Sources:

- Code from *MoqXunitWithDI* in the repository – demonstrates injecting multiple IFooCollection services and async logic in FooLogicImplementergithub.comgithub.com, with unit tests using Moqgithub.comgithub.com.
- Code from *Ttf.WebApi* in the repository – shows a controller using DI with a strategy pattern (IFooCalculatorStrategy with multiple IFooCalculator implementations)github.comgithub.com, plus tests verifying interactions with Moqgithub.comgithub.com.
- Documentation on typical LLM errors with such patterns – LLMs often omit needed dependencies or await calls in heavy-DI scenariosfile-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry, and struggle with factory/strategy indirections (e.g. mocking or invoking the wrong things)file-v3ynhhdenxb5t61zexbjryfile-v3ynhhdenxb5t61zexbjry.