How would you handle API versioning and backward compatibility in microservices ?

Handling API versioning and backward compatibility in microservices requires a dual strategy: explicit versioning of the interface and designing changes to be non-breaking by default.

1. API Versioning Strategies

When you must make breaking changes (removing fields, changing data types), you need a way to serve different versions of your API simultaneously.

Strategy	Implementation	Pros	Cons
URI Versioning	`/api/v1/orders`	Easiest to implement and test. Clients see version explicitly [1].	Clutters the URI. Breaks REST principles (URI should be a resource, not a version) [2].
Header Versioning	`X-API-Version: 1`	Keeps URIs clean. Separates versioning from the resource structure [2] [3].	Harder to test in a browser. Requires clients to configure headers [4].
Content Negotiation	`Accept: application/vnd.company.v1+json`	Purest REST approach. Versions specific representations of a resource [2] [3].	Complex to implement. Can be difficult for simple clients to use [2].

Recommendation: Start with **URI Versioning** for public APIs due to its simplicity and clarity for developers. Use **Header Versioning** for internal microservice-to-microservice communication to keep URLs clean.[4]

2. Ensuring Backward Compatibility

The best way to handle versioning is to avoid breaking changes in the first place. This is often called the "Expand and Contract" pattern or "Additive Changes".[5] [6]

- **Additive Changes (Safe):**
  - **Adding new fields:** Old clients will simply ignore the extra data.[7]
  - **Adding new endpoints:** Existing clients won't even know they exist.
  - **Relaxing constraints:** e.g., making a mandatory field optional.[7]
- **Breaking Changes (Avoid or Version):**

- **Renaming fields:** e.g., changing `userName` to `user_name` breaks clients expecting the old name.[6] [7]
- **Deleting fields/endpoints:** Causes immediate errors for clients relying on them.[6]
- **Tightening constraints:** Making an optional field mandatory will break any client that doesn't send it.[7]

3. Managing the Lifecycle

- **Consumer-Driven Contracts (CDC):** Use tools like **Pact** to define "contracts" between services. This allows you to verify automatically if a change in a provider service will break any of its consumers before you deploy.[8]
- **Postel's Law:** "Be conservative in what you do, be liberal in what you accept from others." Your services should gracefully ignore unknown fields in requests/responses rather than throwing errors.[9]
- **Deprecation Policy:** When you do introduce `v2`, don't kill `v1` immediately. Mark it as deprecated, announce a sunset date, and monitor logs to ensure traffic drops to zero before removing it.[10] [8]

⁂

1. https://www.statsig.com/perspectives/the-importance-of-api-versioning-in-microservices
2. https://www.lonti.com/blog/api-versioning-url-vs-header-vs-media-type-versioning
3. https://www.linkedin.com/pulse/ultimate-guide-microservices-api-versioning-dreamfactory-software-hdise
4. https://www.serverion.com/uncategorized/versioning-strategies-for-microservices-schemas/
5. https://blog.dreamfactory.com/ultimate-guide-to-microservices-api-versioning
6. https://abp.io/community/articles/best-practices-for-designing-backward-compatible-rest-apis-in-a-microservice-solution-for-.net-developers-9rzlb4q6
7. https://www.freecodecamp.org/news/how-to-handle-breaking-changes/
8. https://sealos.io/ai-quick-reference/475-how-do-microservices-handle-versioning
9. https://www.simform.com/blog/microservice-best-practices/
10. https://www.lonti.com/blog/managing-api-changes-and-breaking-changes-in-versioned-apis
11. https://dzone.com/articles/api-versioning-in-microservices-architecture
12. https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
13. https://spring.io/blog/2025/09/16/api-versioning-in-spring
14. https://www.linkedin.com/advice/1/how-can-you-maintain-backward-compatibility-microservices
15. https://dev.to/authress/breaking-up-the-monolith-breaking-changes-14dd