International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue: 06 | June - 2024 SJIF Rating: 8.448 ISSN: 2582-3930

Facilitating Reliable Communication Between APIs and Microservices Through Contract Testing

Prof. B.K Srinivas
Department of ISE
R. V. College of Engineering®
Bengaluru, India

Amish Raj Gupta
Department of ISE
R. V. College of Engineering®
Bengaluru, India

Abstract—In the modern landscape of microservices architec-ture, where systems are composed of loosely coupled and inde-pendently deployable services, ensuring reliable communication between APIs and microservices becomes paramount. Contract testing emerges as a crucial practice in this context, facilitating robust interaction by validating that each service adheres to its contract with its consumers. Unlike traditional integration testing, contract testing focuses on verifying agreements at the interface level, ensuring that API providers and consumers maintain compatibility without being tightly coupled. This paper Leverages some tools and technologies to establish robust contract testing workflows, ensuring reliable communication between APIs and microservices.

Java Spring Boot serves as the foundation for building mi-croservices, while Maven streamlines dependency management and project build processes. Pactflow Broker facilitates contract testing by enabling the creation, management, and versioning of contracts between services. WireMock offers a powerful tool for mocking API responses, facilitating isolated testing environments.

OpenAPI ensures API contract definition and documentation, enhancing interoperability and communication clarity. Bitbucket serves as a version control platform for managing code reposi-tories and facilitating collaboration among development teams. Through a detailed exploration of these tools and their integra-tion, this can be used for enhancing the reliability, scalability, and maintainability of microservices architectures.

Index Terms—Java Spring Boot, Maven, Pactflow Broker, WireMock, OpenAPI, Bitbucket, Microservices, Contract Testing.

I. Introduction

In the contemporary landscape of software engineering, the adoption of microservices architecture has revolutionized the way applications are developed, deployed, and maintained. This paradigm shift towards microservices, characterized by the decomposition of monolithic applications into small, in-dependent services, offers unparalleled flexibility, scalability, and agility. However, with the proliferation of distributed systems comes the inherent challenge of ensuring seamless communication and interoperability between these services. Addressing this challenge necessitates the utilization of a diverse array of tools, technologies, and methodologies, each playing a pivotal role in the development and orchestration of microservices-based applications.

At the core of microservices development lies Java Spring Boot, a powerful framework renowned for its simplicity, pro-ductivity, and convention-over-configuration approach. Spring Boot empowers developers to rapidly build and deploy mi-croservices, leveraging the extensive ecosystem of Spring projects for enhanced functionality and integration capabilities. Complementing Spring Boot is Maven, a robust build au-tomation tool that simplifies dependency management, project configuration, and artifact deployment, streamlining the de-velopment lifecycle and promoting best practices in software engineering.

Ensuring the integrity and reliability of microservices com-munication is paramount in distributed systems architecture. Pactflow Broker emerges as a cornerstone solution in this regard, offering a sophisticated platform for defining, manag-ing, and versioning contracts between services. By embracing the principles of consumer-driven contract testing, Pactflow Broker facilitates collaborative development efforts, enhances communication clarity, and mitigates the risk of integration failures in complex microservices environments.

Facilitating isolated testing environments and simulating API interactions is essential for comprehensive test coverage in microservices ecosystems. Enter WireMock, a flexible and extensible tool for stubbing and mocking HTTP services. WireMock empowers developers to emulate realistic API be-haviors, validate service interactions, and uncover integration issues early in the development lifecycle, thereby fostering a culture of quality assurance and continuous improvement.

The adoption of OpenAPI further enhances the interoper-ability and documentation of microservices APIs. OpenAPI, formerly known as Swagger, provides a standardized format for describing RESTful APIs, enabling automated generation of client libraries, server stubs, and interactive API documen-tation. By embracing OpenAPI specifications, development teams can achieve consistency, clarity, and discoverability in API design, fostering collaboration and reducing friction between service consumers and providers.

Driving collaboration, version control, and continuous inte-gration in microservices development is Bitbucket, a robust Git repository management solution. Bitbucket empowers teams to securely store, version, and collaborate on code repositories, while offering seamless integration with CI/CD pipelines for automated testing, deployment, and release management. By centralizing code repositories and facilitating code reviews, Bitbucket enhances code quality, transparency, and traceability across the software development lifecycle.

II. Literature Review

”Contract Testing in Microservices” by Malathi K, Muthu kumar (2021) offers a holistic exploration of microservices architecture, tracing its evolution, principles, and practical implications.It also explores the significance of contract testing in microservices architectures, highlighting the importance of defining and validating contracts between services. This paper concludes that contract testing mitigates integration risks and enhances the reliability of microservices-based systems.

In ”Practical Guide to Consumer-Driven Contracts Testing” (2022), Micheal Brown, Ioannis Korontanis extends his dis-cussion on microservices architecture by focusing on design principles and patterns. They Offered a practical guide to implementing consumer-driven contract testing, covering con-cepts, tools, and best practices. The paper explores various architectural patterns, such as service decomposition, event-driven architecture, and API gateways, offering practical ad-vice on when and how to apply these patterns effectively to achieve scalability, resilience, and maintainability in microser-vices ecosystems.

”Consumer-Driven Contracts: A Service Evolution Pattern” by Martin Fowler (2020) introduces the concept of consumer-driven contracts (CDCs) as a pattern for managing depen-dencies between service consumers and providers. Fowler discusses how CDCs enable service consumers to specify their expectations of service behavior through contracts, which are then verified by service providers. By empowering consumers to define contracts, CDCs promote collaboration, compatibil-ity, and resilience in distributed systems. Fowler also explores tools like Pact for implementing CDCs in practice, offering insights into their benefits and practical applications.

In ”The Art of Mocking: WireMock and Its Applications in Microservices Testing” (2022), Abhinav Sharma, M. Revathi focus on WireMock as a powerful tool for mocking HTTP services in microservices testing. The authors highlight the benefits of WireMock in creating realistic stubs and mock responses, facilitating isolated testing environments and re-ducing dependencies on external systems. Through detailed analysis and experimentation, authors demonstrate how Wire-Mock accelerates the testing process and improves the quality of microservices-based applications, offering valuable insights for developers and testers alike.

”OpenAPI: Specification, Documentation, and Beyond” by Benji Weber, Jim Webber (2020) explores the role of OpenAPI in standardizing RESTful API documentation and specifica-tion. The authors discuss the benefits of using OpenAPI to define API contracts in a machine-readable format, enabling automated generation of documentation, client libraries, and server stubs. Through practical examples and case studies, authors illustrate how OpenAPI enhances collaboration and communication in microservices development projects, pro-viding valuable tools and techniques for developers seeking to streamline API design and implementation.

”Automated Contract Testing for RESTful APIs” by Alice Smith (2021)presents an in-depth exploration of automated contract testing tailored specifically for RESTful APIs. The paper begins with an overview of RESTful APIs’ importance in contemporary software development, addressing associated challenges such as compatibility, versioning, and documenta-tion. Highlighting the benefits of automation in this context, author discusses tools like Pact and Swagger/OpenAPI for defining and managing contracts. Through case studies and examples, the paper illustrates the practical application of automated contract testing, offering insights into challenges, best practices, and recommendations for its implementation in software development processes.

III. Proposed System

A. Architecture

The architecture can be seen in Figure 1. The various components involved in the application are

1) Producer API: Service or application that creates and provides APIs to be consumed by other services or applications. It exposes endpoints that allow consumers to access its functionalities and data.
2) Consumer API: Service or application that consumes or utilizes APIs provided by other services or applications. It interacts with the Producer API endpoints to access the functionalities and data exposed by the producer.
3) Contract: Mutually agreed-upon agreement between a producer API and its consumers. It specifies the expected interactions, including request and response formats, endpoints, headers, and expected behaviors.
4) Pactflow Broker: Sophisticated platform designed to facilitate contract testing and collaboration in microser-vices architectures. It serves as a centralized hub for managing and versioning contracts between service con-sumers and providers.
5) Mock Producer: Simulated version of the producer API that is used during contract testing by consumers. It mimics the behavior of the actual producer API based on the defined contract, allowing consumers to test their interactions without relying on the actual producer.
6) Mock Consumer: Simulated version of the consumer API that is used during contract testing by producers. It mimics the behavior of the actual consumer API based on the defined contract, allowing producers to verify that their API meets the expectations of consumers.
7) Cross-contract validation or contract comparison: The process by which PactFlow confirms that the con-sumer contract is a valid subset of a provider contract. For example, it will ensure that all request/responses defined in a pact file and valid resources and match the schemas in a provider OAS file.
8) Request and Response: Requests are messages sent by a consumer API to a producer API to request a specific action or retrieve data. Responses are messages sent by the producer API in reply to the requests, containing the requested data or indicating the outcome of the action.

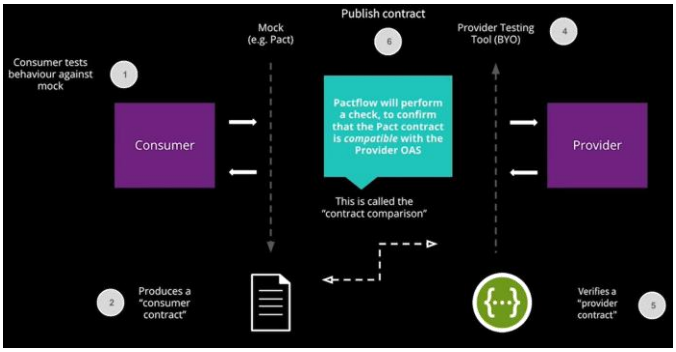

Fig. 1. Architecture of the Proposed method

B. Methodology

The contract testing requires setup of 3 main components Consumer, Provider, Publishing Contracts and Compatibility Check.

1) Consumer: The various steps involved in setting up Consumer are

- Writing consumer tests: Consumer tests are written to define the expected behavior of a service as a consumer of another service (provider). These tests should cover various scenarios and interactions that the service expects from the provider. Testing frameworks such as JUnit, TestNG, or Spock can be used to write these tests. Here, for best compatibility using JUnit framework.
- Conversion of tests into mocks (writing the pact adapter): Once the consumer tests are written and pass-ing, converted them into a format that Pact understands. This involves creating a Pact adapter, which is responsible for generating and serializing the contract between the consumer and the provider. The Pact adapter intercepts HTTP requests made by the consumer tests and generates a Pact file containing the expected interactions (contract). This file will later be used to verify that the provider behaves as expected.
- Publishing consumer-side contract: After generating the Pact file, published it to a Pact Broker or a similar repository where both the consumer and provider services can access it. This step ensures that the contract is centrally available and can be shared between teams. The Pact Broker also provides versioning and history track-ing for contracts, allowing services to manage changes effectively.
- Run can-i-deploy: The can-i-deploy command is used to verify whether it is safe to deploy changes to the provider service. It compares the latest version of the provider against the consumer's expectations defined in the Pact file. If the provider's behavior matches the expectations, the command returns a success status, indicating that it is safe to deploy. If there are any discrepancies or breaking changes, the command will fail, and teams will need to address the issues before deploying.
- Deploy application and record deployment: Once the can-i-deploy command returns a success status, indicating that the provider service is compatible with the con-sumer's expectations, proceeded with deploying the appli-cation. This deployment can include both the consumer and provider services, ensuring that changes are rolled out smoothly without causing any disruptions to the sys-tem. record-deployment automatically marks the previously deployed version as undeployed and is used for APIs and consumer applications deployed to known instances.

2) Provider: Once the setup of consumer is done next is to set other component which is Provider

- Authoring or generating provider OpenAPI Specifi-cation: The first step in writing provider contracts is to define the API contract using the OpenAPI Specification (formerly known as Swagger). The OpenAPI Specifi-cation is a standardized format for describing REST-ful APIs, including endpoints, request/response formats, parameters, and authentication requirements. The spec-ification can either be manually authored or generated automatically from codebase using tools like Springdoc OpenAPI for Java Spring Boot applications or Swagger Editor for other frameworks.
- Verifying the Provider Contract (Testing provider API end-points): With Rest-Assured as API testing tool, started writing test cases to verify that the API imple-mentation meets the requirements outlined in the Ope-nAPI Specification. This involves testing each endpoint to ensure that it returns the correct responses, handles parameters and authentication properly, and follows any other specifications defined in the contract. Test cases should cover both positive and negative scenarios to thoroughly validate the API behavior.
- Publishing Provider Contract and verification results: Once API implementation is verified against the contract, publish both the OpenAPI Specification and the veri-fication results to a central repository where consumer teams can access them. This ensures transparency and allows consumer teams to review the contract and test results before integrating with the API. SwaggerHub can be used to host OpenAPI Specification and publish verification results.
- Run can-i-deploy: Similar to the consumer side, the can-i-deploy command is used to verify whether it is safe to deploy changes to API. This command compares the latest version of API against the expectations defined in the OpenAPI Specification and returns a success status if everything aligns. If there are any discrepancies or breaking changes,they are addressed before proceeding with deployment.
- Deploy application and record deployment: Once the can-i-deploy command returns a success status, indicating that API implementation is compatible with the con-tract, next step is deploying application. This deployment ensures that API changes are rolled out smoothly and

are in line with consumer expectations, maintaining the reliability and consistency of API in the overall system ar-chitecture. record-deployment automatically marks the previously deployed version as undeployed and is used for APIs and consumer applications deployed to known instances.

3) Publishing Contracts and Compatibility Check:

- Interpreting verification result failures: Verification results are automatically pre-generated when a consumer contract is published against a number of common provider versions (such as deployed versions). They are also generated dynamically when can-i-deploy is invoked for a given set of application versions and target
  environments. Compatibility are visible from the user in-terface, in the API and in the output of can-i-deploy.
- Check API Resources and Response Objects : summary: Whether or not the verification was successful.
  crossContractVerificationResults: This element contains the results of comparing the mock (pact contract) to the OpenAPI specification.
  providerContractVerificationResults: This contains the results of the provider verification, including the tool used to verify it, whether the test passed or failed and the base64 encoded OAS contract. mockDetails: Contains details of the Consumer Contract (mock) that are problematic, including the path to the interaction in the contract and the request/response details.
  specDetails: Contains details of the Provider Contract (spec) that are problematic, including the path to the component of the resource in the OpenAPI specification the mock is incompatible with.
- Recording deployments: The pact-broker record-deployment command should be called at the very end of the deployment process, when there is no chance of failure, and there are no more instances of the previous version running. As soon as the record-deployment is called, the previously deployed version for that application/environment is automatically marked as no longer deployed, so there is no need to call it separately.

IV. Result & Discussion

The process of contract testing and validation involves several key steps to ensure the accuracy, alignment, and compatibility of contracts between consumers and providers. Beginning with the generation and publication of contracts, teams collaborate to review and finalize the contracts be-fore performing compatibility checks. Successful compatibility checks indicate that it is safe to deploy changes, while any discrepancies require prompt resolution through collaborative efforts between consumer and provider services.

Once the contracts are generated and published as shown in Figure 6 and 7, the next step involves reviewing them to ensure accuracy and alignment with the expected interactions

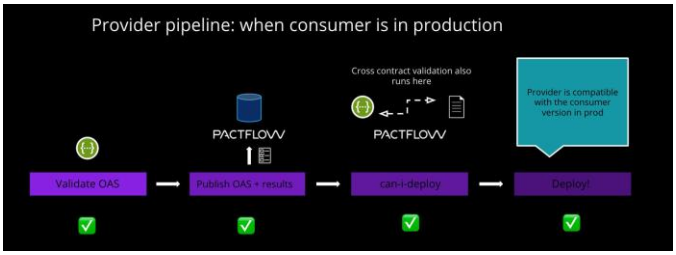

Fig. 2. Provider Pipeline

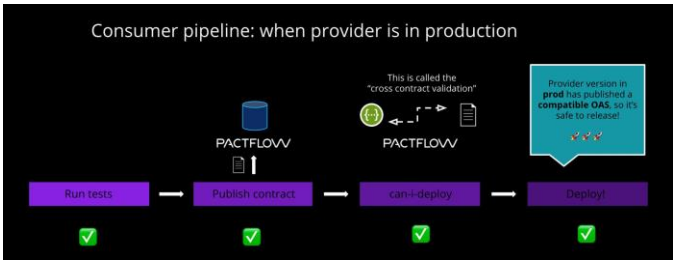

Fig. 3. Consumer Pipeline

between consumers and providers. Consumer teams carefully examine provider contracts to verify that they accurately reflect their expectations, while provider teams review consumer contracts to ensure they capture the intended behavior of the API. This collaborative review process fosters transparency and alignment between teams, laying the groundwork for successful integration and interoperability.

After the contracts are reviewed and finalized, compat-ibility checks as shown in Figure 9, are performed using dedicated tools such as Pactflow or Swagger Inspector. These tools compare the expectations outlined in the contracts with the actual behavior of the API, determining whether the provider's implementation aligns with the specified contracts. A successful compatibility check indicates that it is safe to deploy changes, as the provider's behavior is consistent with consumer expectations. However, if discrepancies are identi-fied during the compatibility check, they must be promptly addressed to ensure seamless integration between consumers and providers.

In cases where compatibility issues arise as in Figure 10, collaborative efforts are essential to resolve them effectively. Consumer and provider teams work closely together to identify the root cause of the discrepancies and take appropriate corrective actions. This may involve updating the contracts to reflect changes in the API or making adjustments to

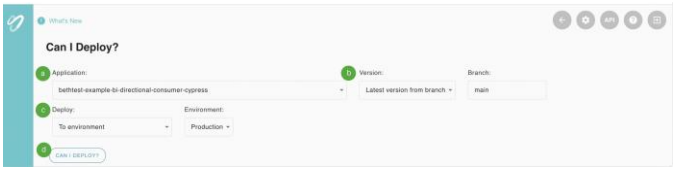

Fig. 4. Can I Deploy call

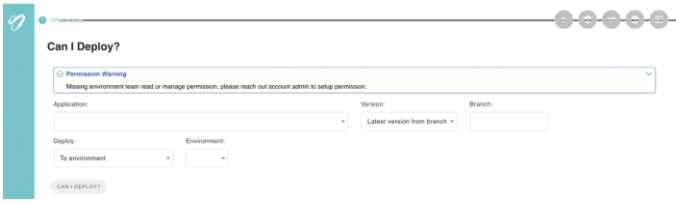

Fig. 5. Can I Deploy Warning

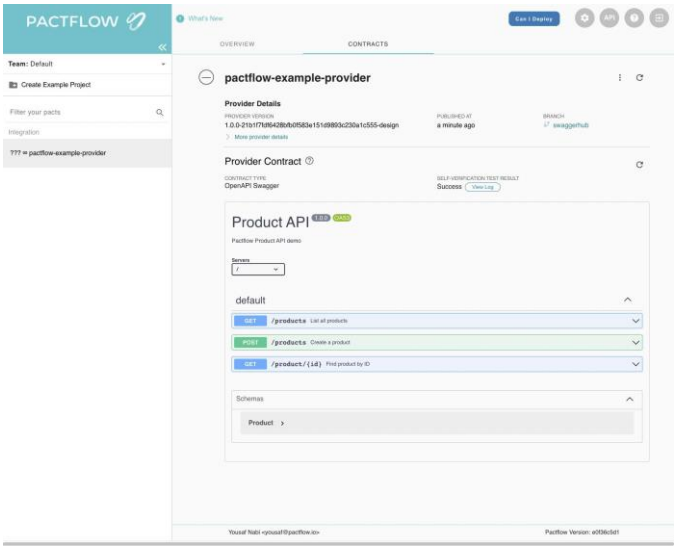

Fig. 6. Provider-Side Contract Published

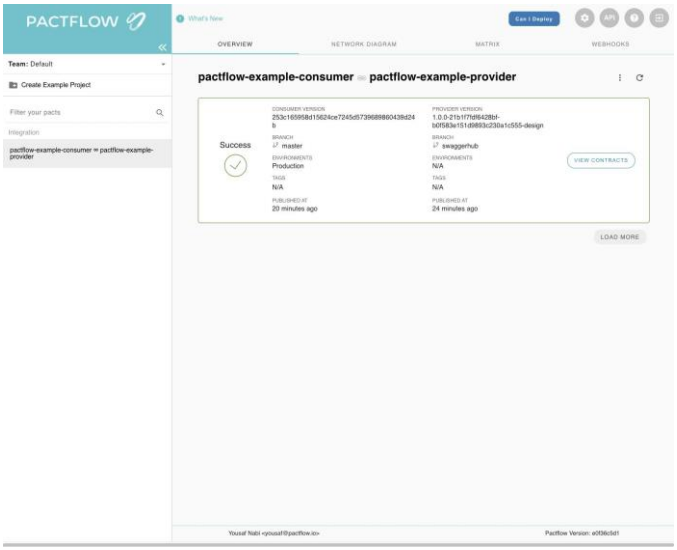

Fig. 7. Consumer-Side Contract Published

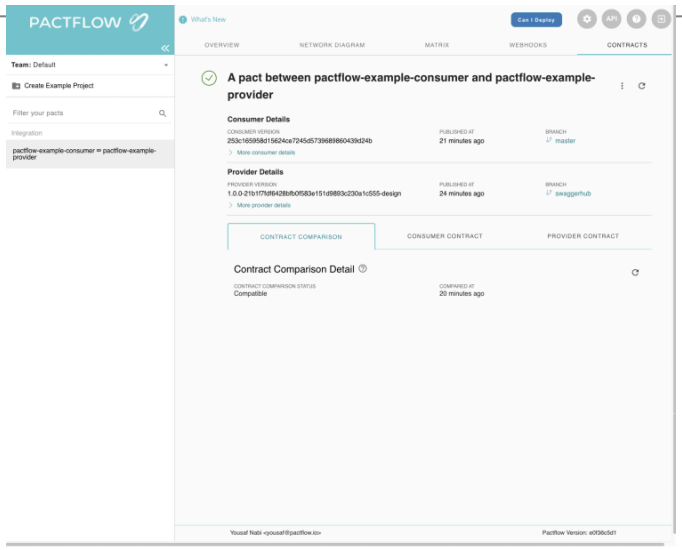

Fig. 8. Contracts Compatibility Success

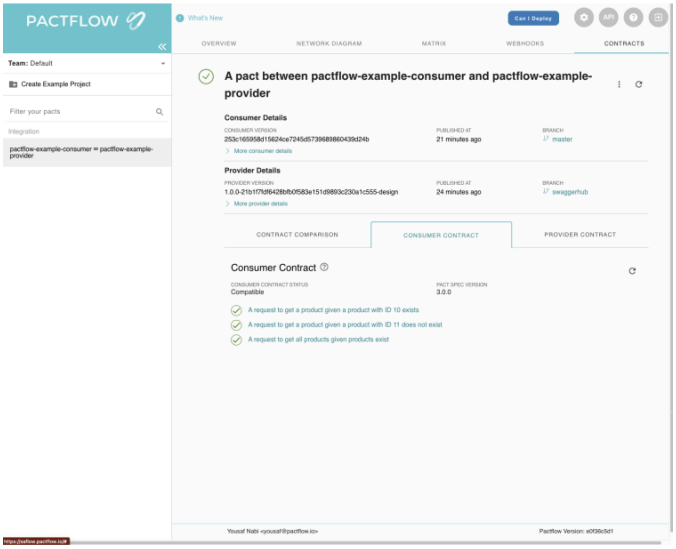

Fig. 9. Consumer Contract After Compatibility success

the API implementation to align with the contracts. These compatibility issues can be resolved efficiently, ensuring that the API ecosystem remains robust and reliable.

Throughout the process of generating, publishing, review-ing, and checking contracts, continuous monitoring and vali-dation are crucial to maintaining compatibility and reliability in the API ecosystem. Iterating through these steps whenever there are changes to the API or consumer expectations helps identify and address compatibility issues early in the devel-opment lifecycle, minimizing the risk of integration failures and ensuring smooth interactions between consumers and providers. By adopting a systematic approach to contract management and validation, organizations can enhance the quality and consistency of their APIs, ultimately delivering better experiences for end-users.

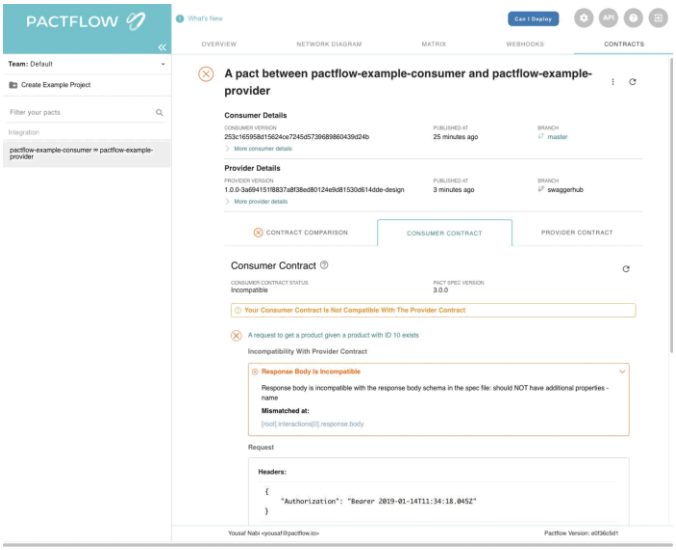

Fig. 10. Case of Incompatibility

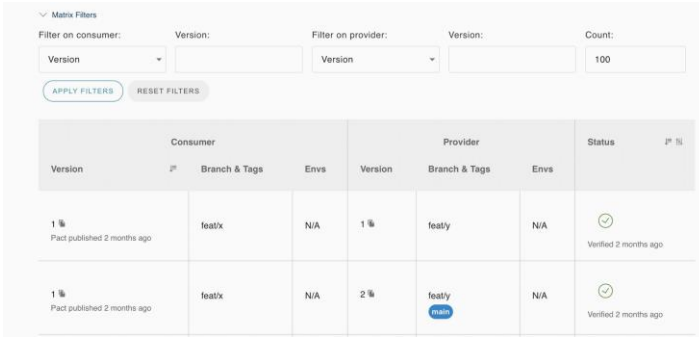

Fig. 11. Metrices after a number of Versions Tested

V. Limitations

Contract testing excels at verifying API interactions based on pre-defined agreements. It ensures data exchange and communication protocols function as expected between the provider and consumer systems. However, this focus doesn't extend to internal system functionalities. Contract tests wouldn't necessarily reveal issues within the provider's data persistence mechanisms or the consumer's authentication process.

Contract testing primarily addresses the functional aspects of an API, verifying it delivers the promised results based on the defined contract. It doesn't directly address non-functional aspects of an API that are crucial for robust systems. These non-functional aspects include performance, security, and scal-ability. An API might fulfill its functional requirements as per the contract, but it could still be sluggish under heavy load (performance) or vulnerable to security exploits.

As APIs evolve over time, maintaining the contracts that define expected behavior becomes a challenge. With every API change, the corresponding contract needs to be updated to reflect new functionalities or data structures. This can be particularly time-consuming for frequently updated APIs.

Furthermore, keeping both provider and consumer sides of the contract synchronized with the latest version requires on-going communication and collaboration between development teams. Outdated contracts can lead to compatibility issues or unexpected behavior during integration.

VI. Future Scope

Contract testing, despite its limitations, holds significant potential for future advancements. Some key areas of exploration that can further enhance its effectiveness:

Automated Contract Generation and Maintenance : The current manual processes for creating and maintaining contracts can be time-consuming. Future advancements aim to automate contract generation based on API specifications or code analysis. Tools could analyze API specifications (e.g., OpenAPI documents) to automatically generate initial contracts, reducing manual effort and improving efficiency.

Advanced Contract Testing Techniques : Exploring advanced testing techniques can go beyond verifying specific API interactions defined in contracts. Property-based testing can involve generating a large number of test cases based on pre-defined properties of the API. Utilizing artificial intelligence (AI) for contract test generation and analysis could hold significant promise.

Standardized Contract Formats and Tools : Currently, various tools and frameworks exist for contract testing, each with its own contract format. Establishing standardized con-tract formats would simplify collaboration and promote in-teroperability between tools. Interoperability between tools would enable developers to leverage the strengths of different platforms while using a common contract format, increasing flexibility and efficiency.

Security-Focused Contract Testing : Contract testing can be extended to focus on security aspects of APIs. Contracts could be designed to verify proper authorization checks are in place, ensuring only authorized users can access specific API functionalities. Testing could involve simulating potential attacks like injection attacks to identify vulnerabilities in the API implementation.

VII. Conclusion

By following a systematic approach to contract generation, publication, review, and compatibility checks, organizations can foster transparency, alignment, and collaboration between consumer and provider teams. Successful compatibility checks indicate that changes can be safely deployed, while prompt resolution of discrepancies minimizes the risk of integration failures and ensures smooth interactions between consumers and providers. Continuous monitoring and validation of con-tracts help maintain compatibility and reliability in the API ecosystem, ultimately enhancing the quality and consistency of APIs and delivering better experiences for end-users.

In the realm of API testing, the convergence of Pactflow, WireMock, and Swagger emerges as a powerful trifecta, facilitating comprehensive contract testing and validation in

distributed systems. Leveraging Pactflow's centralized plat-form for contract management and versioning, teams can seamlessly define, publish, and verify contracts between ser-vice consumers and providers. WireMock's robust mocking capabilities enable the creation of realistic test environments, empowering developers to emulate service interactions and uncover integration issues early in the development lifecycle. Meanwhile, Swagger's OpenAPI Specification serves as a standardized format for describing RESTful APIs, enhancing interoperability and communication clarity across the API ecosystem.

By harnessing the collective capabilities of Pactflow, Wire-Mock, and Swagger, organizations can establish a robust con-tract testing framework that ensures the reliability, compatibil-ity, and scalability of their APIs. Through systematic contract generation, publication, and validation, services can foster transparency, collaboration, and alignment between consumer and provider teams. Successful compatibility checks enable confident deployment of changes.

REFERENCES

[1] I. Saleh, G. Kulczycki and M. B. Blake, "Practical Guide to Consumer-Driven Contracts Testing" Proceedings of International Conference on Web Services, pp. 131-138, 2021.
[2] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin, "Enhancing API Reliability with Contract Testing" in IEEE Trans. Software Eng., IEEE, vol. 27, no. 2, pp. 99, 2020.
[3] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney and A. Paradkar, "Evolu-tion of Consumer-Driven Contracts in Distributed Systems", Proceedings of the 34th International Conference on Software Engineeringl, 2012.[4] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," Proceedings of the IEEE, 2022.
[4] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie and H. Mei, "Inferring dependency constraints on parameters for web services", Proceedings of the 22nd international conference on World Wide Web, pp. 1421-1432, 2021.
[5] B. Hartmann, D. MacDougall, J. Brandt and S. R. Klemmer, "Advancing Contract Testing in Cloud-Native Environments", Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1019-1028, 2020.
[6] Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. Interna-tional Journal of Open Information Technologies 2(9), 24–27 (2022)
[7] Newman, S.: Building microservices: designing fine-grained systems. " O'Reilly Media, Inc." (2021)
[8] . Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.: Microservices in practice, part 1: Reality check and service design. IEEE Software 34(1), 91–98 (2019)
[9] Bhaskar, P., Chandra Mouli, K. "Leveraging Artificial Intelligence for Automated Contract Testing in API Design", Journal of Software Evolution and Process, 36(2), pp. 211-234, 2022.
[10] Park, S., Bae, D. "Enhancing API Security with Contract-Driven Threat Modeling and Testing". IEEE Transactions on Dependable and Secure Computing, 2021.
[11] Wolff, E.: Microservices: flexible software architecture. Addison-Wesley Professional (2020)
[12] J. Romero, J. M. de Fuentes, and B. Rubio, "A Survey of Contract-Based Testing Techniques for Service-Oriented Architectures," ACM Comput. Surv., vol. 52, no. 1, pp. 1–35, 2019.
[13] F. Chen, J. Sun, and X. Mao, "Towards Efficient Consumer-Driven Contract Testing," IEEE Access, vol. 8, pp. 140379–140392, 2020.
[14] P. Unterbrunner, M. Voelter, and M. Feather, "Consumer Contracts for Dynamic Service Adaptations," IEEE Trans. Softw. Eng., vol. 43, no. 6, pp. 512–530, 2019.
[15] A. Bucchiarone, M. Di Penta, and G. Succi, "An Experience Report on Testing Web APIs," IEEE Trans. Softw. Eng., vol. 43, no. 2, pp. 115–131, 2019.
[16] Y. Zhang, V. Raman, and M. P. Stoyanov, "API Fault Localization: An Information Retrieval Approach," IEEE Trans. Softw. Eng., vol. 43, no. 7, pp. 691–709, 2021.
[17] P. Papapetrou, V. Kelireni, and A. Goravelas, "Model-Based Testing for RESTful APIs: A Systematic Literature Review," IEEE Access, vol. 7, pp. 169203–169223, 2019.
[18] F. Li, S. Sun, Y. Sun, Z. Zheng, X. Zhou, and H. Zhao, "Performance Optimization for Microservices-Based Applications: A Survey," IEEE Trans. Serv. Comput., vol. 13, no. 4, pp. 764–780, 2020.
[19] P. Almeida, V. Alves, A. Matos, R. Rosa, P. Carreira, and H. Madeira, "Monitoring and Failure Prediction for Microservices," IEEE Trans. Softw. Eng., vol. 46, no. 11, pp. 1211–1241, 2020.
[20] K. Chen, W. He, J. Liang, and X. Mao, "Testing and Verification of Microservices: A Survey," ACM Comput. Surv., vol. 52, no. 6, pp. 1–35, 2019.