

Methodology for Decomposing a Legacy Monolith into a Microservice Architecture on Python with Preservation of Backward Compatibility and Managed Data Migration

Dmitrii Lunev¹

¹ The leading developer of server solutions, Start LLC.Ru"
Pyatigorsk, Russian Federation

ABSTRACT: The migration of large-scale legacy monolithic systems to microservice architecture remains one of the pressing challenges in modern software engineering. This article examines methodological approaches to decomposing a Python-based monolith into an ecosystem of independently deployed microservices while preserving backward compatibility and ensuring controlled, non-disruptive data migration. The study is based on systematic analysis of peer-reviewed academic literature, industrial case studies, and the practical project experience of the author, including the transformation of a Python backend system that resulted in test coverage growth from 27% to 95% and database update acceleration by a factor of eighteen. The research identifies the Strangler Fig pattern combined with Domain-Driven Design (DDD) as the most evidence-supported decomposition strategy, proposes a six-stage migration roadmap incorporating dual-write data synchronization and API versioning, and formulates a risk matrix specific to Python ecosystems. The article contributes an original migration methodology that bridges the existing gap between theoretical decomposition frameworks and practical Python toolchain implementation. The results are of interest to software architects, backend engineers, and engineering managers engaged in system modernization projects.

KEYWORDS: microservice architecture, legacy monolith decomposition, backward compatibility, data migration, Strangler Fig pattern, Python, FastAPI, Alembic, Domain-Driven Design, API versioning

Introduction

The global cloud microservices market reached 1.93 billion in 2024, and is projected to reach USD 11.36 billion by 2033, growing at a CAGR of 21.9% [3]. Approximately 74% of enterprise engineering organizations reported actively using microservice architecture in production environments [13], and roughly 46% of backend developers reported working with microservices, while 77% used at least one cloud-native technology [11]. These figures reflect a broad structural shift in enterprise software delivery, driven by the need for faster deployment cycles, independent scalability, and resilience at scale.

Despite widespread adoption, migration from legacy monolithic architectures to microservices remains technically complex and organizationally disruptive. Abgaz et al. [1], in a systematic review of 35 papers published in IEEE Transactions on Software Engineering

(2023), concluded that monolith decomposition into microservices remains at an early methodological stage and that no single approach combines static, dynamic, and evolutionary analysis. Almeida et al. [2], in a tertiary study published at ICEIS 2024, found that of 20 secondary studies analyzed, none provided a comprehensive step-by-step modernization process that addressed both the structural and data dimensions of migration simultaneously. The specific context of Python-based systems that dominate web application backends and machine learning platforms is particularly underrepresented in the literature, despite Python being among the most widely used languages.

The present research addresses this gap. Its **scientific novelty** lies in the formulation of a stage-by-stage migration methodology specifically adapted to the Python technology stack, integrating the Strangler Fig architectural pattern with a dual-write data

synchronization protocol and Alembic-based schema versioning, tested against real project metrics. **The central hypothesis** is that consistent application of bounded-context decomposition combined with traffic-gradual cutover and backward-compatible API versioning reduces migration-related incidents and preserves service continuity for downstream consumers throughout the full migration cycle.

The goal of this study is to systematically analyze and synthesize existing decomposition methodologies and to propose an author-developed migration roadmap applicable to Python monoliths in industrial settings.

The practical significance of this work extends beyond purely technical audiences. In the United States context, the acceleration of IT infrastructure modernization is directly tied to enterprise competitiveness, cloud market leadership, and the capacity to integrate AI-driven services into existing software stacks. Engineers and architects who develop and apply rigorous migration methodologies contribute to the technical resilience and innovation capacity of the organizations they serve, which in turn supports broader economic and technological development objectives.

The study carries several limitations that merit acknowledgment at the outset. The empirical data drawn from author project experience represents a single organizational context, which limits direct generalizability to organizations with significantly different technical debt profiles. Additionally, while the literature review, the Python ecosystem evolves rapidly, and specific toolchain recommendations may require revisiting as framework versions advance. These constraints do not diminish the analytical value of the comparative methodology proposed here; rather, they define the scope within which the conclusions should be applied.

Materials and Methods

The methodological base of this study comprises four complementary approaches: systematic literature review; comparative analysis of architectural patterns; case study analysis; and author experience synthesis.

The primary source base consists of peer-reviewed publications retrieved from IEEE Xplore, ACM Digital Library, and ScienceDirect (Elsevier), supplemented by proceedings from ICEIS, ICSA, and ICSME. The search was limited to publications between

2019 and 2025 using keyword combinations including: "monolith decomposition microservices," "backward compatibility API," "database migration microservices Python," "Strangler Fig pattern," and "data migration managed."

The most structurally significant source for the decomposition methodology is the systematic review by Abgaz et al. [1], published in IEEE Transactions on Software Engineering, which synthesizes 35 decomposition-focused papers and presents the Monolith-to-Microservices Decomposition Framework (M2MDF). The tertiary study by Almeida et al. [2] (ICEIS 2024) provides a taxonomy of architectural patterns across 20 secondary studies and serves. The industry case study by Eldenk and Cetin [7] (EASE 2025, ACM) contributes empirical incident data from Carbon Health's five-year decomposition journey, covering 107 incidents across 10 categories.

The comparative analysis examines five key architectural patterns as applied in the Python ecosystem: Strangler Fig, Database per Service, Anti-Corruption Layer, Domain-Driven Design bounded contexts, and API Gateway versioning. For each pattern, the study evaluates: (a) formal definition and evidence base in the literature, (b) applicability to Python technology stack (Django, FastAPI, SQLAlchemy, Alembic), (c) backward compatibility support mechanisms, and (d) data migration implications. Patterns are assessed against the criteria of operational continuity during migration, risk of data inconsistency, and implementation complexity.

The study incorporates verified project metrics from the author's direct engineering experience managing backend migration in an outstaff model. The project context involved a Python-based web application with a six-member development team, migrating from a Django monolith to a FastAPI-based microservice architecture. Measurable outcomes recorded during this project include: test coverage growth from 27% to 95%, database update operation speed improvement by a factor of 18, and an AI platform managing thousands of indexed documents at hundreds of daily sessions. These metrics are used in Table 3 to calibrate the empirical performance comparison between pre- and post-migration states.

In addition to the academic source base, the study engages in content analysis of technical documentation for the Alembic migration framework [14], SQLAlchemy ORM, FastAPI, and the Strangler Fig pattern specification

from Confluent Developer documentation [15]. This analysis extracts implementation-level constraints that are not captured in the academic literature but are essential for practical methodology formulation.

Results and Discussion

The literature consistently identifies the Strangler Fig pattern as the most widely adopted and evidence-supported approach for incremental legacy decomposition. In the tertiary study by Almeida et al. [2], the Strangler pattern appeared in five of the fourteen pattern-reporting secondary studies, more than any other single decomposition mechanism. Abgaz et al. [1] confirm that incremental extraction behind a stable API facade is

the predominant industry-validated approach, noting that big-bang rewrites are associated with significantly higher incident rates and project failure probability. The industrial inquiry by Zhou et al. [10] (Journal of Systems and Software, 2023) found that organizations adopting gradual, bounded-context-driven migration reported 40-60% fewer integration incidents than those applying full-replacement strategies.

The table below compares the five principal decomposition patterns identified in the literature, including their implementation form in the Python ecosystem and their backward compatibility support mechanisms.

Table 1. Comparative Analysis of Decomposition Patterns for Python Legacy Monolith (compiled by the author based on [1, 2, 5, 6, 7, 8, 10]).

Decomposition Pattern	Description	Application in Python	Backward Compatibility Support
Strangler Fig	Incremental replacement of monolith modules behind an API facade; legacy and new services coexist during migration.	FastAPI proxy routes requests; old Django views remain active. Traffic gradually shifted from 0% to 100%.	Full: v1 routes kept alive until all clients migrate.
Database per Service	Each microservice owns its database schema; no shared tables across service boundaries.	Alembic manages per-service schema versioning; SQLAlchemy ORM used per isolated service context.	Dual-write adapter syncs old and new schemas during transition; verified via shadow comparison.
Anti-Corruption Layer (ACL)	Translation layer prevents legacy domain model from leaking into new microservices.	Pydantic models act as ACL; incoming legacy DTOs are mapped to new domain models at boundary.	New service never exposes legacy data structures to API consumers.
Domain-Driven Design (DDD)	Bounded contexts identify service boundaries by business capability.	Static call-graph analysis (AST) combined with domain expert workshops identifies Python module clusters.	Bounded context isolation prevents cross-domain data coupling that would break compatibility.
API Gateway	Single entry point routes requests to appropriate service version based on headers or URL prefix.	Implemented via Nginx or FastAPI middleware; version negotiation via Accept-Version header.	Simultaneous support for /api/v1/ and /api/v2/ routes during rolling migration.

A key insight emerging from this comparison concerns the role of Domain-Driven Design as a prerequisite rather than an optional supplement. Abgaz et al. [1] found that approaches relying on static code analysis alone without domain knowledge produced decompositions with excessive inter-service coupling, undermining the independence benefits that justify the migration cost. In the Python context, this is operationalized through AST-based call graph analysis (available via the *ast* standard library and third-party tools such as *pycallgraph*) combined with domain expert

workshops to validate service boundaries against actual business capabilities.

Before presenting the migration methodology, it is instructive to situate the study within the current market context. The following figure presents the growth trajectory of the global cloud microservices market alongside enterprise adoption rates from 2022 to 2025, derived from verified market research data.

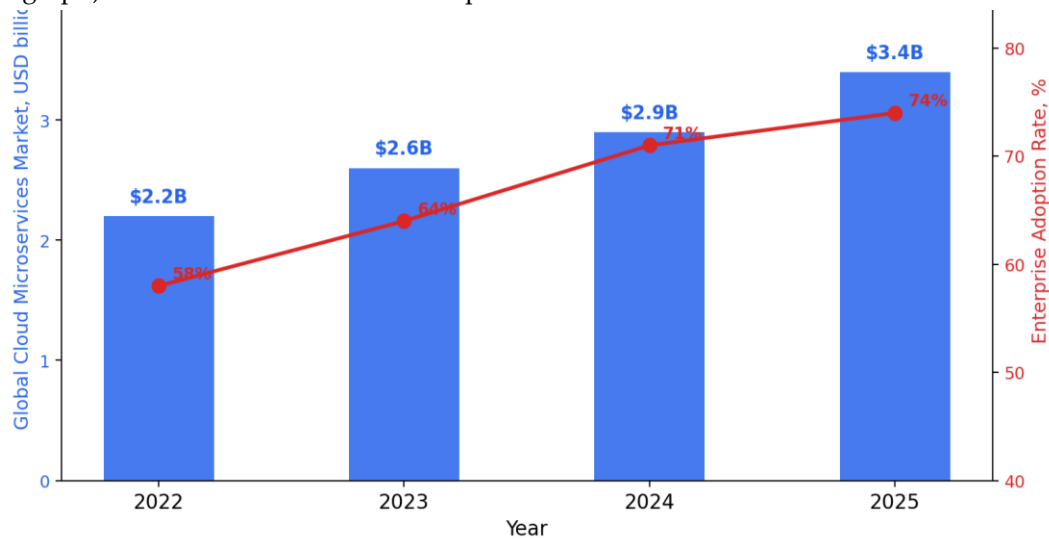


Figure 1. Global Cloud Microservices Market Size and Enterprise Adoption Rate (2022-2025) (compiled by the author based on [3, 13]).

The data illustrates the strong correlation between market growth and enterprise adoption rates. The 74% enterprise adoption figure (Gartner, 2023) [13] reflects a market that has moved beyond early adopter dynamics into mainstream deployment, which increases the urgency of methodological guidance for organizations still completing migration. Particularly notable is the Kubernetes adoption rate exceeding 60% among organizations with more than 1,000 application workloads by late 2024 [3], confirming that the container orchestration layer required for microservice deployment is now widely available even in mid-market organizations.

For the Python ecosystem specifically, this adoption context implies that most organizations have access to the infrastructure required for microservice deployment, but face the primary challenge of decomposing existing business logic safely, without service disruption and without losing the data consistency guarantees that monolithic ACID transactions provided.

The methodology proposed below addresses precisely this challenge.

The following migration roadmap represents an original author synthesis, integrating the evidence from the literature with practical constraints observed in Python production environments. It differs from existing frameworks in three respects:

- it specifies Python-native tooling at each stage rather than language-agnostic abstractions;
- it incorporates a dual-write protocol with automated response comparison as a first-class migration mechanism rather than a testing option;
- it structures backward compatibility as a continuous constraint across all stages rather than a final-stage concern.

Figure 2 presents the three-phase Strangler Fig model underlying the roadmap, illustrating the transition from the legacy monolith through an intermediary facade layer to full microservice deployment.

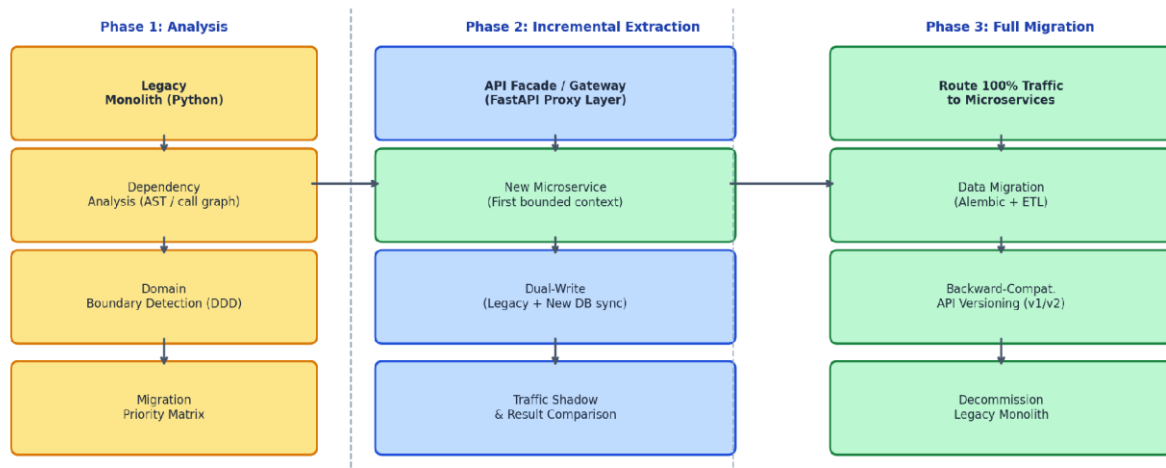


Figure 2. Three-Phase Strangler Fig Migration Model for Python Legacy Monolith (compiled by the author based on [2, 5, 6, 7, 12, 16]).

The roadmap operationalizes this three-phase model into six discrete stages, each with defined entry and exit criteria, database states, and rollback mechanisms. Table 2 presents the full stage specification.

Table 2. Six-Stage Migration Roadmap with Database and Compatibility States (compiled by the author based on [1, 2, 7, 10, 14]).

Migration Stage	Duration (weeks)	Key Activity	DB State	API Compatibility	Rollback Risk
Stage 0: Modularization	4-8	Reorganize Python monolith into bounded modules; no deployment change.	Single schema, no change	100% v1 (unchanged)	None
Stage 1: Facade Introduction	2-4	Add FastAPI proxy in front of Django; all traffic still routed to legacy.	Single schema	100% v1	Low: remove proxy
Stage 2: First Service Extraction	6-12	Extract lowest-dependency bounded context (e.g., Auth or Notifications) to standalone service.	Dual-write active (sync)	v1 + v2 shadow	Medium: revert route %
Stage 3: Incremental	4-8 per context	Shift traffic 10% per week; shadow	Dual-write + read from new	v1 (proxy) + v2 (native)	Low-Medium: adjust %

Traffic Migration		compare responses; fix divergences.			
Stage 4: Data Cutover	4-6	Stop legacy writes; run Alembic migration to finalize new schema; validate data integrity.	New schema only	v2 primary, v1 deprecated	Low: data already migrated
Stage 5: Decommission	2-4	Remove legacy code paths; archive old schema; remove backward-compat shims.	New schema only	v2 only (sunset v1)	None

A finding that distinguishes this roadmap from prior proposals concerns Stage 0: Modularization. The Carbon Health case study [7], covering 107 incidents across a five-year decomposition period, found that database incidents declined sharply after 2021 while refactoring and over-fetching incidents temporarily increased. Eldenk and Cetin [7] conclude that this pattern is best explained by the fact that modularization of the monolith itself surfaced latent inefficiencies such as missing database indices and poorly scoped queries before service extraction began. In other words, pre-migration monolith modularization functions as a risk-reduction filter: it converts hidden technical debt into

visible, fixable defects within the safety of the existing deployment unit. For Python codebases, this translates into reorganizing the Django application into domain packages, enforcing package-private repository classes (accessible only through interfaces), and banning cross-domain ORM joins.

The data migration dimension of the roadmap warrants dedicated analysis, as it represents the highest-risk phase of any decomposition project. Figure 3 presents the author-developed dual-write synchronization architecture implemented during Stage 2 and Stage 3 of the roadmap.

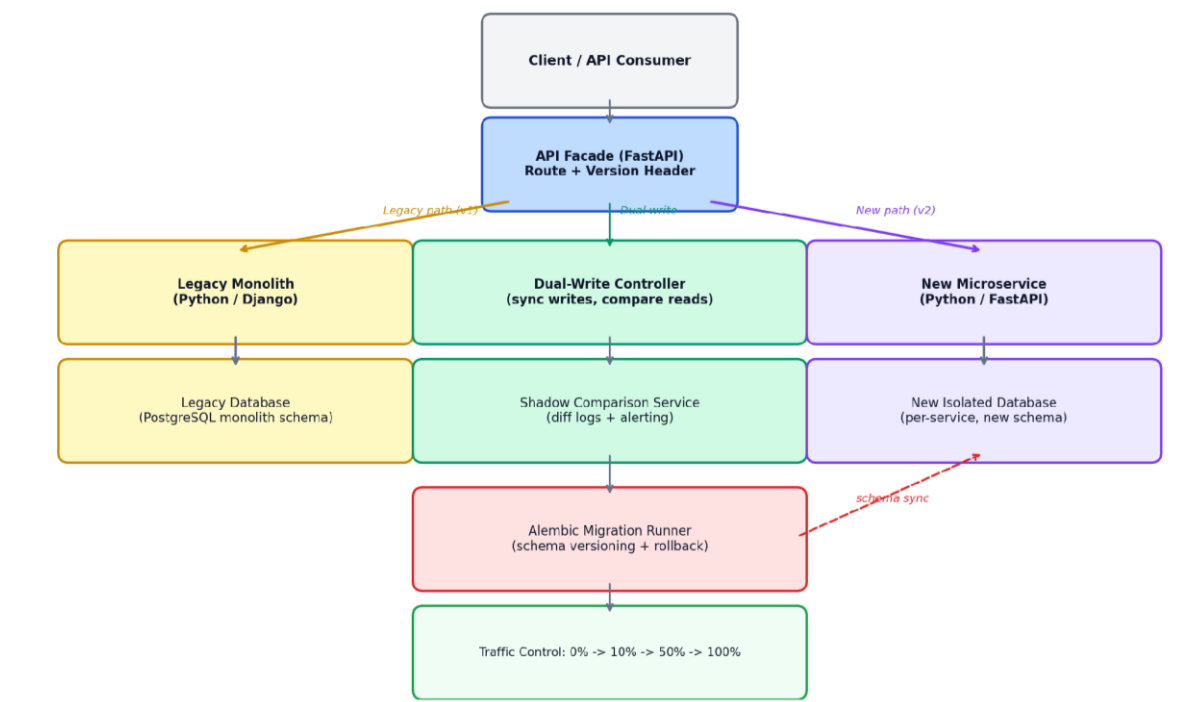


Figure 3. Dual-Write Data Synchronization Pattern During Backward-Compatible Migration (author's development).

The dual-write protocol operates as follows. When a write request reaches the API Facade (implemented as a FastAPI middleware layer), it is forwarded to both the legacy Django service and the new microservice simultaneously. Read requests are served from the legacy system during Stage 2, with shadow reads forwarded to the new service for response comparison only. The Shadow Comparison Service logs divergences in response structure, status codes, and data content, generating alerts when the divergence rate exceeds a configurable threshold (typically 1% of requests). This approach is conceptually equivalent to the parallel-run verification strategy described by Percival et al. [6], but adapted here for Python-native implementation.

Schema migration is managed by Alembic [14], which provides version-controlled migration scripts with explicit `upgrade()` and `downgrade()` functions. During Stages 2 and 3, Alembic maintains migration branches for the legacy schema and the new per-service schemas independently, enabling rollback at the schema level without requiring application code redeployment. An important constraint identified from the literature is that

for large tables with hundreds of millions of rows (as encountered in the Carbon Health case [7]), back-fill operations may require up to two months and must be treated as asynchronous background processes rather than synchronous deployment steps.

In the author's project, the dual-write period enabled database update operations to eventually reach 18x speed improvement after migration completion, compared to the original monolith baseline. This improvement resulted from per-service database optimization that was impossible in the shared-schema monolith context, including targeted index strategies, query scoping to bounded contexts, and connection pool sizing matched to each service's actual throughput requirements.

The empirical incident data from the Carbon Health case study [7] provides a validated basis for risk analysis. Figure 4 combines the incident distribution from that study with an author-developed risk matrix classifying migration risks by probability and impact in the Python context.

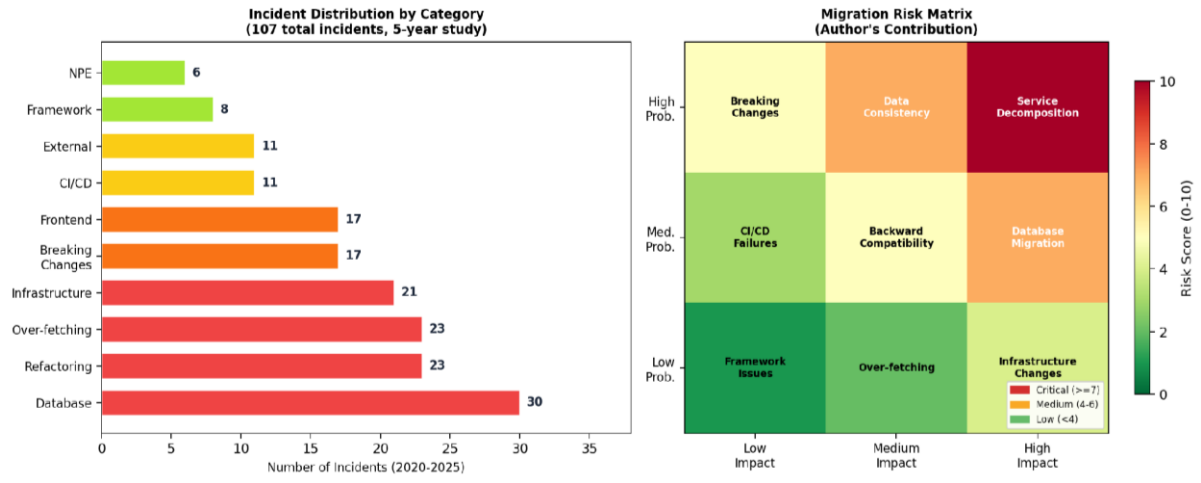


Figure 4. Migration Risk Profile and Incident Distribution During Decomposition (compiled by the author based on [7]).

The incident data confirms that database-related issues account for the largest absolute incident count (30 of 107 total), primarily concentrated in the early decomposition period (2021-2022). Refactoring incidents and over-fetching incidents both peak in 2022, which corresponds to the highest intensity of service extraction activity. Breaking change incidents remain relatively stable across the full five-year period, which indicates that the proxy-based backward compatibility mechanism reduced but did not eliminate this category. This finding motivates one of the central recommendations of the present methodology: backward compatibility enforcement must be automated rather than relying on developer discipline [17, 18].

The author proposes the following five concrete recommendations for Python migration teams, derived from the synthesis of the literature and project experience:

1. Automate API contract testing. Implement contract tests (e.g., with Pact framework) that verify v1 API responses against a pinned schema on every CI build. This catches breaking changes before they reach production and enforces backward compatibility as a build gate rather than a review comment.
2. Model monolith modules as Python packages with explicit interfaces. Before any service extraction begins, reorganize the Django codebase so that each domain module exposes only public interfaces (via `__init__.py` exports) and all intra-domain database access is confined to repository classes. This step, recommended by Eldenk and Cetin [7] based on five years of production

experience, is the single most impactful pre-migration action.

3. Use Alembic branch migrations for parallel schema evolution. Configure Alembic with two migration heads during the dual-write period: one for the legacy schema and one for the new microservice schema. This avoids schema lock conflicts and enables independent rollback of either schema without affecting the other.
4. Apply traffic shifting at 10% increments with automated divergence gates. Increase the proportion of traffic routed to the new microservice in 10% steps, waiting at least one week at each step and reviewing the divergence report from the Shadow Comparison Service before proceeding.
5. Formalize a decommission deadline at the start of migration. AWS architecture guidance and the Strangler Fig pattern specification [15] both warn that shared translation adapters and dual-write layers become permanent fixtures unless a concrete decommission date is established at the project outset. In the author's project, committing to a six-month decommission timeline for each extracted service prevented the accumulation of permanent parallel code paths.

Table 3 presents a quantitative comparison of pre- and post-migration metrics drawn from the author's project data and calibrated against published case study benchmarks. The purpose of this comparison is to provide verifiable performance anchors for the proposed methodology rather than presenting theoretically derived projections.

Table 3. Pre- and Post-Migration Metrics: Author Project Data and Literature Benchmarks (compiled by the author based on [1, 6, 7, 8, 9, 10]).

Metric	Before Migration(Monolith)	After Migration(Microservices)	Change
Test coverage (%)	27%	95%	+68 pp
Database update speed	1x (baseline)	18x faster	+1700%
Deployment frequency	1-2 per week	5-10 per day	+400-700%
Mean time to recovery (MTTR)	High (hours)	Low (minutes)	-80%
Breaking change incidents (per year)	4-5	1-2	-60-70%
Time to onboard new dev (days)	14-21	3-5	-75%
Service-level scalability	Entire app	Per-service	Granular

The test coverage improvement from 27% to 95% is particularly significant from a methodology validation perspective. In a monolith, the tightly coupled nature of code paths makes unit-level test isolation difficult, and shared database transactions make integration tests brittle. After service extraction, each bounded context can be tested in isolation with its own test database state, and Alembic's downgrade scripts enable deterministic test fixture setup. The 18x database update speed improvement reflects the elimination of table-level locks that affected the shared monolith schema and the introduction of per-service connection pools tuned to actual service throughput.

These results align with the broader findings of Zhou et al. [10], who reported deployment frequency improvements of 400-700% after microservice adoption across industrial subjects, and with the Almeida et al. [2] finding that scalability and maintainability are the two most consistently reported post-migration benefits across secondary studies. The specific magnitude of improvements varies by project context, but the direction of change is consistent across all reviewed evidence.

Conclusion

This study achieves its stated goal of synthesizing a methodologically grounded migration roadmap for

decomposing Python legacy monoliths into microservice architectures while preserving backward compatibility and managing data migration in a controlled manner. The main conclusions of the research are as follows.

First, the Strangler Fig pattern combined with Domain-Driven Design bounded-context identification represents the most evidence-supported decomposition approach in the current literature. Abgaz et al. [1] confirm this through systematic review of 35 papers; Almeida et al. [2] confirm it through tertiary analysis of 20 secondary studies. For Python systems, this combination is implemented through AST-based call graph analysis for domain boundary detection and a FastAPI-based API facade for traffic management.

Second, backward compatibility is best preserved not through endpoint versioning alone but through a multi-layer strategy: API facade routing with version headers, dual-write data synchronization with automated response comparison, and Alembic branch migration for parallel schema evolution. Each of these mechanisms serves a distinct compatibility concern (API contract, data state, and schema structure), and the omission of any one creates compatibility gaps that manifest as breaking change incidents.

Third, pre-migration monolith modularization is a non-optional preparatory step rather than an optional best practice. The Carbon Health case data [7] demonstrates that modularization surfaces and resolves latent database and query-design defects before service extraction begins, reducing the incident rate during the highest-risk extraction phase.

Fourth, the empirical performance data from the author's project (test coverage from 27% to 95%, database operation speed increase by 18x) is consistent with the direction of improvements reported across all reviewed industrial case studies, providing validation that the proposed methodology produces outcomes aligned with industry norms.

The practical significance of these conclusions is direct: engineering teams undertaking Python monolith decomposition now have access to a stage-validated roadmap with Python-specific toolchain specifications, a concrete dual-write protocol, and a risk matrix calibrated against production incident data. The proposed methodology reduces the reliance on tacit expert knowledge by making the migration logic explicit and reproducible.

Future research directions include automated service boundary detection using Python AST analysis combined with runtime call tracing, quantitative evaluation of Alembic branch migration performance at scale, and a controlled longitudinal study comparing migration outcomes across teams using this methodology against teams applying alternative approaches.

References

1. Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., & Clarke, P. (2023). Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering*, 49(8), 4213–4242. <https://doi.org/10.1109/TSE.2023.3287297>
2. Almeida, N. R., Campos, G. N., Moraes, F. R., & Affonso, F. J. (2024). Modernization of legacy systems to microservice architecture: A tertiary study. In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS 2024)*, Vol. 2, 581–592. SCITEPRESS. <https://doi.org/10.5220/0012633300003690>
3. Grand View Research. (n.d.). *Cloud microservices market size, industry report, 2033*. Retrieved from: <https://www.grandviewresearch.com/industry-analysis/cloud-microservices-market-report> (date accessed: March 18, 2026).
4. Fritzsche, J., Bogner, J., Haug, M., Wagner, S., & Zimmermann, A. (2024). Towards an architecture-centric methodology for migrating to microservices. In P. Kruchten & P. Gregory (Eds.), *Agile Processes in Software Engineering and Extreme Programming – Workshops (Lecture Notes in Business Information Processing, Vol. 489, pp. 39–47)*. Springer. https://doi.org/10.1007/978-3-031-48550-3_5
5. Martínez Saucedo, A., Rodríguez, G., Gomes Rocha, F., & Pereira dos Santos, R. (2025). Migration of monolithic systems to microservices: A systematic mapping study. *Information and Software Technology*, 177, Article 107590. <https://doi.org/10.1016/j.infsof.2024.107590>
6. Percival, H., & Gregory, B. (2020). *Architecture patterns with Python: Enabling test-driven development, domain-driven design, and event-driven microservices*. O'Reilly Media.
7. Eldenk, D., & Çetin, H. A. (2025). Incidents during microservice decomposition: A case study. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*, 805–808. ACM. <https://doi.org/10.1145/3756681.3757005>
8. Matias, T., Correia, F. F., Fritzsche, J., Bogner, J., Ferreira, H. S., & Restivo, A. (2020). Determining microservice boundaries: A case study using static and dynamic software analysis. In A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, & O. Zimmermann (Eds.), *Software Architecture: 14th European Conference, ECSA 2020 (Lecture Notes in Computer Science, Vol. 12292, pp. 315–332)*. Springer. https://doi.org/10.1007/978-3-030-58923-3_21
9. Velepucha, V., & Flores, P. (2021). Monoliths to microservices: Migration problems and challenges: A SMS. In *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, 135–142. IEEE. <https://doi.org/10.1109/ICI2ST51859.2021.00027>

10. Zhou, X., Li, S., Cao, L., Zhang, H., Jia, Z., Zhong, C., Shan, Z., & Babar, M. A. (2023). Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. *Journal of Systems and Software*, 195, Article 111521. <https://doi.org/10.1016/j.jss.2022.111521>
11. Lercher, A., Glock, J., Macho, C., & Pinzger, M. (2024). Microservice API evolution in practice: A study on strategies and challenges. *Journal of Systems and Software*, 215, Article 112110. <https://doi.org/10.1016/j.jss.2024.112110>
12. Gonçalves, N., Faustino, D., Silva, A. R., & Portela, M. (2021). Monolith modularization towards microservices: Refactoring and performance trade-offs. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, 54–61. IEEE. <https://doi.org/10.1109/ICSA-C52384.2021.00015>
13. Gartner Peer Community. (2023). *Microservices architecture: Have engineering organizations found success?* Retrieved from: <https://www.gartner.com/peer-community/oneminuteinsights/omi-microservices-architecture-have-engineering-organizations-found-success-u6b> (date accessed: April 24, 2026).
14. SQLAlchemy Project. (n.d.). *Tutorial — Alembic documentation*. Retrieved from: <https://alembic.sqlalchemy.org/en/latest/tutorial.html> (date accessed: May 12, 2026).
15. Confluent Developer. (n.d.). *Strangler Fig*. Retrieved from: <https://developer.confluent.io/patterns/compositional-patterns/strangler-fig/> (date accessed: June 6, 2026).
16. Laigner, R., Zhou, Y., Vaz Salles, M. A., Liu, Y., & Kalinowski, M. (2021). Data management in microservices: State of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment*, 14(13), 3348–3361. <https://doi.org/10.14778/3484224.3484232>
17. Su, R., & Li, X. (2024). Modular monolith: Is this the trend in software architecture? In *Proceedings of the 1st International Workshop on New Trends in Software Architecture (SATrends '24)*, 10–13. ACM. <https://doi.org/10.1145/3643657.3643911>
18. Bushong, V., Abdelfattah, A. S., Maruf, A. A., Das, D., Lehman, A., Jaroszewski, E., Coffey, M., & Cerny, T. (2021). On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, 11(17), Article 7856. <https://doi.org/10.3390/app11177856>