

**FROM XAMARIN TO .NET MAUI:
A RISK-WEIGHTED MIGRATION FRAMEWORK FOR LARGE ENTERPRISE
MOBILE PORTFOLIOS****Mallikarjuna Yadav Bachu**
Lead Mobile Developer, USA**ABSTRACT**

Large enterprises operating mobile application portfolios built on Xamarin face a strategic inflection point following Microsoft's introduction of .NET Multi-platform App UI, announced at the Microsoft Build developer conference in May 2021 as the evolution of Xamarin.Forms within a unified .NET runtime. For organizations maintaining dozens or hundreds of Xamarin applications, migration to the new framework cannot be executed as a single uniform project, since individual applications vary enormously in size, architectural complexity, third party dependency exposure, and business criticality. This research article proposes a risk weighted migration framework that segments a large enterprise mobile portfolio into prioritized migration cohorts based on a structured, multi factor risk score, rather than migrating applications in an arbitrary or purely chronological order. The framework combines a portfolio inventory methodology, a risk factor taxonomy covering user interface complexity, custom renderer usage, third party library compatibility, platform specific code exposure, team skill readiness, and test coverage, and a scoring model that converts these factors into a single weighted risk index per application. The article further presents migration pattern options, a phased roadmap template, an effort estimation model, and an illustrative simulation applying the framework to a representative sample portfolio, alongside extensive supporting reference tables intended for direct reuse in enterprise migration planning exercises.

- Enterprises with mature Xamarin portfolios have been reported in industry surveys to operate between 20 and 300 or more distinct Xamarin applications across business units, depending on organizational size.
- Custom renderer usage, a primary source of migration friction discussed in this article, has been identified in comparable enterprise Xamarin.Forms codebases in 30 to 60 percent of applications with nontrivial custom user interface requirements.
- Illustrative migration effort estimates presented in this article range from approximately 2 person-weeks for a minimal, low complexity application to over 40 person-weeks for a large, highly customized application with extensive platform specific code.
- The risk weighted framework presented in this article uses 7 primary risk factors, each scored on a 1 to 5 scale and combined through a weighted index ranging from 7 to 35 points per application.

1. INTRODUCTION

Xamarin has served as a primary cross platform mobile development framework for enterprise organizations since Microsoft's acquisition of the technology in 2016, allowing development teams to share a substantial portion of application logic and, through Xamarin.Forms, user interface code across iOS, Android, and Windows targets from a shared C# and .NET codebase. Over the years following that acquisition, large enterprises accumulated substantial mobile application portfolios built on this foundation, spanning customer facing applications, internal line of business tools, and specialized applications for field service, logistics, and retail operations.

At Microsoft Build in May 2021, Microsoft announced .NET Multi-platform App UI, positioned as the next evolution of Xamarin.Forms and part of a broader strategy to unify the .NET platform across web, desktop, cloud, and mobile workloads under a single base class library and tooling experience aligned with the then upcoming .NET 6 release. For enterprises with substantial existing Xamarin investment, this announcement raised an immediate and practical planning question: how should a portfolio of dozens or hundreds of existing Xamarin applications be prioritized, sequenced, and resourced for eventual migration, given that migrating all applications simultaneously is rarely feasible from either a budget or engineering capacity standpoint.

1.1 The Portfolio Prioritization Problem

Most published migration guidance addresses the technical mechanics of migrating a single application, covering topics such as project file conversion, namespace changes, and the replacement of the custom renderer extensibility model with the new handler based architecture. This technical guidance, while necessary, does not address the distinct and equally important portfolio management question of sequencing across many applications with differing risk profiles. A large enterprise cannot treat migration purely as an engineering task; it must also be treated as a portfolio risk management exercise, allocating limited migration engineering capacity toward the applications and sequencing that minimize overall organizational risk and disruption.

- A portfolio level view is necessary because migrating high risk, high complexity applications first, before organizational migration expertise has matured, has been associated in comparable enterprise technology transitions with higher defect rates and schedule overruns.
- Conversely, deferring all migration until Xamarin reaches end of support risks a compressed, high pressure migration window across the full remaining portfolio simultaneously, a scenario this framework is explicitly designed to help organizations avoid.

1.2 Objectives and Structure of the Article

This article has five objectives. First, it summarizes the architectural relationship between Xamarin, Xamarin.Forms, and .NET MAUI sufficient to ground the risk factors discussed later. Second, it presents a portfolio assessment methodology, including an inventory approach and a seven factor risk taxonomy. Third, it details a weighted scoring model that converts portfolio inventory data into a prioritized migration sequence. Fourth, it surveys migration patterns, phased roadmap structure, effort estimation, and testing strategy applicable once an application has been selected for migration. Fifth, it presents an illustrative simulation applying the framework to a representative sample portfolio and discusses governance, cost, and limitations.

2. Literature Review

The migration framework proposed in this article draws on three distinct bodies of prior work: software migration and legacy modernization literature, portfolio and risk management standards, and Xamarin and .NET MAUI specific technical documentation.

Legacy modernization literature has long recognized that large scale technical migrations benefit from incremental rather than wholesale replacement strategies. The strangler fig pattern, describing a gradual replacement approach in which new functionality progressively surrounds and eventually replaces a legacy system, has been widely referenced as a risk reducing alternative to a single large cutover, and directly informs the phased migration patterns discussed in Section 8 of this article. Literature on migrating from monolithic to distributed architectures has similarly emphasized incremental extraction guided by explicit risk and complexity assessment, a philosophy this article adapts from service level migration to application portfolio level migration.

The concept of technical debt, originally introduced to describe the accumulating cost of expedient but suboptimal software design decisions, provides a useful conceptual bridge between the migration risk factors catalogued in Section 6 and the broader body of software maintainability literature. Applications exhibiting extensive custom renderer usage, heavy platform specific code branching, or thin automated test coverage can be understood as carrying elevated technical debt specifically with respect to framework portability, a framing that helps explain why such applications require disproportionately higher migration effort relative to their raw size, as reflected in the effort adjustment factors presented in Section 10.

Portfolio and risk management standards provide the structural foundation for the scoring model presented in Section 5. Established risk management guidance defines risk in terms of likelihood and impact combined into a composite assessment, a structure this article's weighted scoring model follows directly by combining a per factor severity score with a factor specific weight reflecting its typical impact on migration outcomes. Project portfolio management literature further emphasizes prioritization frameworks that allocate constrained resources across a portfolio of initiatives based on relative value and risk, a pattern directly analogous to allocating migration engineering capacity across an application portfolio.

Technical documentation and early guidance published around the .NET MAUI announcement, alongside pre-existing Xamarin.Forms architectural documentation, provides the specific technical basis for the risk factors catalogued in Section 6, including the shift from the custom renderer extensibility model to the handler based architecture and the broader ambition of a single project structure spanning multiple platform targets. Because .NET MAUI was newly announced at the time of this article's publication, the technical migration guidance available at publication remains preliminary relative to the mature, multi year body of Xamarin.Forms documentation and community practice that preceded it, a gap this article's risk weighted approach is specifically designed to manage.

Table 1. Architectural Comparison of Xamarin, Xamarin.Forms, and .NET MAUI as Announced in May 2021

Attribute	Xamarin (Xamarin.iOS / Xamarin.Android)	Xamarin.Forms	.NET MAUI (as announced, 2021)
Underlying runtime	Mono runtime	Mono runtime	.NET 6 unified runtime (planned)
UI code sharing model	Platform-specific native UI code	Shared XAML/C# UI with native rendering	Shared XAML/C# UI with native rendering
Extensibility model	Direct platform APIs	Custom renderers	Handlers (new architecture)
Project structure	Separate projects per platform	Shared project plus per- platform head projects	Single project targeting multiple platforms (planned)
Desktop platform support	Not applicable	Limited (UWP)	Windows and macOS (Mac Catalyst) planned
Primary IDE tooling	Visual Studio, Visual Studio for Mac	Visual Studio, Visual Studio for Mac	Visual Studio, Visual Studio for Mac (planned)

.NET MAUI details reflect the framework as announced and in early preview at the time of this article; final released capabilities may evolve as the platform matures toward general availability.

3. Background: Xamarin, Xamarin.Forms, and .NET MAUI

3.1 Xamarin Native Versus Xamarin.Forms

Xamarin, in its original native form, allowed developers to write application logic in C# while building user interfaces using platform specific APIs for each target platform, sharing business logic but not presentation code. Xamarin.Forms, introduced subsequently, added a shared, abstracted user interface layer expressed in Extensible Application Markup Language and C#, rendered to native platform controls through an extensibility mechanism known as custom renderers when default rendering behavior required customization. The large majority of enterprise Xamarin applications built in the years preceding this article's publication were built on Xamarin.Forms rather than the fully native approach, given its substantially higher code sharing potential across platforms.

3.2 The .NET MAUI Announcement and .NET Unification Strategy

.NET Multi-platform App UI was announced as part of a broader .NET unification strategy under which Xamarin, Mono, and the traditional .NET Framework and .NET Core lineages would converge onto a single .NET 6 base class library and runtime, targeting a unified developer experience across web, desktop, cloud, and mobile workloads. For mobile specifically, .NET MAUI was positioned as the direct successor to Xamarin.Forms, replacing the custom renderer extensibility model with a new handler based architecture intended to reduce per platform rendering overhead and simplify the process of supporting additional platform targets beyond the historical iOS, Android, and Windows scope, extending to macOS through Mac Catalyst.

3.3 Key Differences Relevant to Migration Risk

Several architectural differences between Xamarin.Forms and .NET MAUI as announced directly drive the risk factors catalogued later in this article, most notably the change from custom renderers to handlers, the consolidation of previously separate per platform head projects into a single project structure, and the dependency on the .NET 6 unified runtime rather than the legacy Mono based Xamarin runtime, which has downstream implications for third party library compatibility discussed in Section 7.

4. Why Enterprises Need a Risk-Weighted Migration Framework

A large enterprise mobile portfolio is rarely homogeneous. Applications differ in age, in the degree to which they rely on custom renderers or platform specific code, in the currency and compatibility of their third party dependencies, and in the availability of automated test coverage that can validate correct behavior after migration. Treating every application as equally ready for migration, or migrating applications purely in order of business unit request rather than technical readiness, risks concentrating migration defects and schedule overruns in exactly the applications least equipped to absorb them.

4.1 Consequences of an Unweighted Approach

- Migrating a high complexity application early, before team migration expertise has matured through experience on simpler applications, has been associated in comparable enterprise technology transitions with elevated defect rates during initial migration waves.
- Migrating applications with poor existing automated test coverage without first improving that coverage increases the risk that behavioral regressions introduced during migration are not caught before production release.
- Migrating applications with heavy third party dependency exposure without first confirming compatibility status, discussed in Section 7, risks discovering blocking compatibility gaps mid migration rather than during planning.

4.2 Principles of the Risk-Weighted Approach

- Assess before sequencing. Every application in scope should receive a structured risk assessment, described in Section 5, before being assigned to a migration wave.
- Sequence from low to high risk within capacity constraints. Early migration waves should favor lower risk applications, building organizational migration expertise and reusable patterns before attempting the highest risk applications.
- Treat business criticality as a separate dimension from technical risk. A low technical risk, high business criticality application may still warrant careful scheduling around change freeze periods even though its technical migration risk is low.
- Revisit scores periodically. Risk scores should be refreshed as the .NET MAUI platform matures and as third party library compatibility improves over time, rather than treated as a one time, static assessment.

4.3 Organizational Scale Considerations

The specific portfolio sizes, wave counts, and durations discussed throughout this article assume a large enterprise context, broadly defined as an organization managing 50 or more distinct mobile applications under some form of centralized portfolio governance. Organizations below this scale can generally apply the same underlying risk taxonomy and scoring model described in Section 5, but may reasonably compress the five wave structure presented in Section 9 into two or three waves, or in some cases a single wave preceded by a short pilot, given the reduced coordination overhead of a smaller application population.

- Mid-sized portfolios of roughly 15 to 50 applications commonly compress the five-wave structure in Table 10 into 3 waves while retaining the same underlying risk-based sequencing logic.
- Very large portfolios exceeding 200 applications, more common among the largest global enterprises, may warrant splitting the framework's governance structure, described in Section 13, into multiple parallel migration streams organized by business unit or platform team, each following the same shared risk taxonomy but managed against a locally scoped roadmap.

5. Portfolio Assessment Methodology**5.1 Inventory and Classification**

The first step in the framework is constructing a complete inventory of applications in scope, capturing attributes needed to support the risk scoring model described in Section 5.3.

Table 2. Portfolio Inventory Attributes Collected for Each Application

Inventory Attribute	Description	Typical Source
Application name and business owner	Identifies the application and the accountable business stakeholder	Application portfolio management system
Lines of code (approximate)	Rough size indicator used in effort estimation	Source code repository analysis
Number of distinct screens or pages	Alternative size indicator, often more intuitive to business stakeholders than lines of code	Application inventory or manual review
Custom renderer count	Count of custom renderer classes, a primary migration risk factor	Source code repository analysis
Third-party NuGet package list	Full dependency list used in compatibility assessment per Section 7	Project dependency manifest

Platform-specific code volume	Estimated proportion of code using platform-specific APIs directly	Source code repository analysis
Automated test coverage percentage	Existing unit and UI test coverage, used in risk scoring	Continuous integration test coverage reports
Business criticality tier	Organizational classification of business importance, independent of technical risk	Application portfolio management system
Release cadence	Frequency of production releases, relevant to migration scheduling	Release management system

A complete inventory across all attributes is recommended before risk scoring begins, though organizations may begin with a subset and expand coverage iteratively.

5.2 Risk Factor Taxonomy

The framework defines seven primary risk factors, each addressing a distinct dimension of migration difficulty or uncertainty.

Table 3. Risk Factor Taxonomy and Illustrative Weighting

Risk Factor	What It Measures	Weight (of 100 percent)
User interface complexity	Volume and intricacy of custom visual design and layout	15 percent
Custom renderer dependency	Extent of reliance on the custom renderer extensibility model	20 percent
Third-party library compatibility	Availability and maturity of MAUI-compatible dependency versions	20 percent
Platform-specific code exposure	Volume of direct platform API usage outside shared abstractions	15 percent
Automated test coverage	Existing regression test coverage available to validate migration correctness	15 percent
Team migration skill readiness	Team familiarity with .NET MAUI architecture and migration tooling	10 percent
Release cadence and change freeze exposure	Scheduling constraints imposed by business-critical release windows	5 percent

Weightings are illustrative defaults; organizations should adjust weights to reflect their own historical migration experience and risk tolerance once available.

5.3 Scoring Model

Each risk factor is scored on a five point scale for every application, with 1 representing minimal risk and 5 representing severe risk. The weighted risk index for an application is computed as the sum, across all seven factors, of each factor's raw score multiplied by its weight, then scaled to a common reporting range.

Table 4. Five-Point Risk Scoring Scale Applied to Each Risk Factor

Score	Qualitative Label	General Description
1	Minimal	Negligible migration friction expected; suitable for early migration waves
2	Low	Minor, well-understood migration friction with established mitigation approaches
3	Moderate	Meaningful migration effort required; achievable within a standard migration wave

4	High	Substantial migration friction; recommended for later waves after organizational experience matures
5	Severe	Major blocking risk requiring dedicated remediation or vendor engagement before migration

The same five-point scale is applied consistently across all seven risk factors in Table 3 to allow direct aggregation into a single weighted index.

6. Risk Factor Deep Dive

6.1 User Interface Complexity

User interface complexity captures the volume and intricacy of custom visual design elements, animations, and layout structures within an application, since more complex interfaces generally require more careful visual regression testing after migration to the new handler based rendering architecture.

Table 5. User Interface Complexity Scoring Indicators

Indicator	Score 1-2 (Low)	Score 3 (Moderate)	Score 4-5 (High to Severe)
Distinct screens or pages	Fewer than 15	15 to 40	More than 40
Custom animations	None or minimal	A few isolated animations	Extensive custom animation throughout
Complex layout patterns	Standard list and form layouts	Some nested or conditional layouts	Extensive custom layout logic
Platform-specific visual adaptation	Minimal platform-specific styling	Moderate platform-specific styling	Extensive platform-specific visual branching

6.2 Custom Renderer Dependency

Because .NET MAUI replaces the custom renderer extensibility model with handlers, applications with extensive custom renderer usage face a structurally different migration path than applications relying primarily on default, unmodified control rendering.

Table 6. Custom Renderer Count as a Migration Risk Indicator

Custom Renderer Count	Score	Typical Migration Implication
0 to 2	1	Minimal handler migration effort; largely default control usage
3 to 8	2	Moderate handler migration effort concentrated in a small number of controls
9 to 20	3	Meaningful handler migration effort across a range of custom controls
21 to 40	4	Substantial handler migration effort requiring dedicated planning
More than 40	5	Severe handler migration effort; consider partial redesign rather than direct port

6.3 Third-Party Library Compatibility

Third party library compatibility, addressed in detail in Section 7, is scored based on the proportion of an application's dependencies confirmed to have a compatible .NET MAUI or .NET 6 targeted version available at assessment time, given that .NET MAUI was newly announced and third party ecosystem support remained in early stages at the time of this article's publication.

6.4 Platform-Specific Code Exposure

Applications that invoke platform specific application programming interfaces directly, for example for advanced camera control, specialized hardware integration, or platform specific background processing behavior, require additional validation effort during migration, since the underlying platform interop mechanisms are subject to change as part of the broader .NET 6 unification described in Section 3.2.

6.5 Automated Test Coverage

Automated test coverage functions as a risk mitigant rather than a risk source; low coverage increases migration risk because behavioral regressions are less likely to be caught before production release, while high coverage reduces risk by providing a reliable safety net during migration.

Table 7. Automated Test Coverage as a Migration Risk Indicator

Automated Test Coverage	Score	Recommended Action Before Migration
Above 70 percent	1	Proceed with migration using existing test suite as regression safety net
50 to 70 percent	2	Proceed with minor targeted coverage improvements to critical paths
30 to 50 percent	3	Invest in coverage improvement for core business workflows before migration
10 to 30 percent	4	Substantial coverage investment recommended before migration begins
Below 10 percent	5	Coverage investment or comprehensive manual test planning required as a prerequisite

6.6 Team Migration Skill Readiness

Because .NET MAUI represents a new framework with limited historical community practice at the time of this article's publication, team familiarity with the framework's handler architecture, single project structure, and migration tooling is itself a risk factor, mitigated through training, pilot projects, and knowledge sharing across migration waves.

6.7 Release Cadence and Change Freeze Exposure

Applications subject to frequent business critical releases or extended seasonal change freeze periods, such as retail applications during peak shopping seasons, require migration scheduling that respects these constraints independent of the application's underlying technical risk score.

6.8 Composite Risk Interactions

Individual risk factors do not operate entirely independently, and organizations applying the scoring model should watch for compounding interactions among factors rather than treating the weighted sum as a complete description of migration difficulty. An application scoring moderately on both custom renderer dependency and third party library compatibility, for example, often experiences greater combined difficulty than the arithmetic sum of the two factors would suggest, because renderer replacement work frequently surfaces additional dependency compatibility issues that were not visible during initial assessment.

- Applications scoring 3 or higher on both custom renderer dependency and third party library compatibility simultaneously have been associated in comparable enterprise migrations with effort overruns exceeding the Table 12 adjustment factor by an additional 15 to 30 percent.
- Organizations applying the framework are encouraged to flag any application with two or more risk factors scoring 4 or higher for manual review by an experienced migration lead before final wave assignment, rather than relying solely on the automated weighted index.

7. Dependency and Library Compatibility Assessment

Third party library compatibility represents one of the highest weighted risk factors in the framework, given that .NET MAUI's dependency on the unified .NET 6 runtime, as opposed to the legacy Mono based Xamarin runtime,

requires many third party package maintainers to publish updated, compatible package versions before existing applications can adopt them.

Table 8. Illustrative Third-Party Dependency Compatibility Landscape

Dependency Category	Typical Compatibility Status (as of Mid-2021)	Assessment Recommendation
Core .NET base class library packages	Generally compatible given shared .NET 6 foundation	Low priority for detailed review
Popular open source MVVM and navigation libraries	Mixed; many actively updating for MAUI compatibility	Review release notes and community migration guidance per package
UI control and charting libraries	Early stage; handler-based equivalents still emerging	High priority for early compatibility testing
Platform-specific plugin libraries (camera, sensors, biometrics)	Variable; dependent on individual maintainer roadmap	Direct vendor or maintainer outreach recommended
Internal, organization-authored shared libraries	Depends entirely on internal team migration timeline	Coordinate internal library migration ahead of dependent application migration
Analytics and crash reporting SDKs	Mixed; major vendors typically prioritize new framework support	Confirm vendor roadmap commitment before migration wave assignment

Compatibility status reflects the early, evolving state of the .NET MAUI ecosystem at the time of this article's publication and should be reverified against current package documentation at assessment time.

7.1 Compatibility Assessment Process

- Step 1: Extract the full dependency manifest for each application in the portfolio inventory described in Section 5.1.
- Step 2: Classify each dependency against the categories in Table 8, checking maintainer documentation or release notes for stated .NET MAUI or .NET 6 compatibility.
- Step 3: For dependencies without confirmed compatibility, identify whether an alternative package, an internal replacement, or direct maintainer engagement is the most viable path.
- Step 4: Aggregate the proportion of an application's dependencies with confirmed compatibility into the risk score described in Section 6.3.
- Step 5: Track compatibility status centrally across the portfolio, since a dependency confirmed incompatible for one application is very likely shared across other applications in the same portfolio.

7.2 Vendor Engagement Playbook

For dependencies identified as incompatible or with unclear compatibility status, direct engagement with the maintaining vendor or open source project is often more productive than waiting passively for compatibility to emerge, particularly for widely shared dependencies identified through the analysis described above.

Table 8a. Illustrative Vendor and Open Source Maintainer Engagement Playbook

Engagement Step	Objective	Typical Timeline
Identify maintainer support channel	Locate the appropriate issue tracker, support contract, or community forum	1 to 2 days
Submit compatibility inquiry with application context	Obtain a stated roadmap commitment or workaround guidance	1 to 3 weeks for initial response
Evaluate interim workaround or fork option	Determine whether a temporary mitigation unblocks migration scheduling	1 to 2 weeks
Escalate through commercial support contract if applicable	Accelerate roadmap commitment for business-critical dependencies	2 to 6 weeks depending on vendor
Document final resolution path in shared dependency tracker	Ensure resolution benefits all applications sharing the dependency per Section 7.3	Ongoing

7.3 Shared Dependency Leverage

Because large enterprise portfolios frequently share common internal libraries and popular third party packages across many applications, resolving a compatibility gap for a widely used shared dependency delivers risk reduction benefit across every application that depends on it, making shared dependency remediation a high leverage early investment within the overall migration program.

- Portfolio level dependency analysis in comparable enterprise environments has found that a relatively small number of shared dependencies, often fewer than 20 distinct packages, account for the majority of dependency relationships across a large application portfolio.
- Prioritizing compatibility remediation for the most widely shared dependencies before beginning application level migration waves has been associated with meaningfully reduced rework across subsequent migration waves in comparable large scale technology transitions.

8. Migration Patterns and Strategies

Once an application's risk profile has been assessed, the organization must select a migration pattern appropriate to that risk profile and to the application's operational constraints.

Table 9. Comparison of Migration Pattern Options

Pattern	Description	Best Suited To	Relative Risk
Direct in-place migration	Convert the existing project directly to the new project structure and framework APIs	Low complexity, low risk applications with strong test coverage	Low to moderate
Strangler fig incremental migration	Introduce new screens or modules in the new framework while legacy screens remain temporarily unconverted	Larger applications where a full cutover is impractical within one release cycle	Moderate
Parallel run migration	Build and validate the migrated application alongside the legacy application before cutover, comparing behavior	High business criticality applications where regression risk must be minimized	Moderate to high implementation effort, lower cutover risk
Rewrite with modernization	Use the migration as an opportunity for broader architectural modernization beyond a like-for-like port	Legacy applications with substantial accumulated technical debt independent of the framework migration itself	High

Pattern selection should be informed by, but is distinct from, the risk score computed in Section 5.3; a high risk score may favor a strangler fig or parallel run pattern specifically to manage that elevated risk.

8.1 Selecting a Pattern Based on Risk Profile

- Applications scoring in the minimal to low overall risk range are typically well suited to direct in-place migration, since the underlying migration mechanics are expected to be straightforward.
- Applications scoring in the moderate range, particularly those with elevated custom renderer or dependency compatibility scores, often benefit from a strangler fig approach that isolates migration risk to a subset of screens or modules at a time.
- Applications scoring in the high to severe range, especially those with high business criticality, are commonly best served by a parallel run approach that allows extensive validation before cutover, despite the additional implementation effort of maintaining two versions temporarily.

9. Phased Migration Roadmap

Sequencing an entire portfolio's migration typically spans multiple waves, each incorporating lessons learned from the prior wave.

Table 10. Illustrative Five-Wave Portfolio Migration Roadmap

Wave	Target Risk Profile	Typical Portfolio Share	Primary Objective
Wave 0: Pilot	Minimal risk, low business criticality	2 to 5 percent of portfolio	Build initial team migration experience and reusable tooling
Wave 1: Early adoption	Minimal to low risk	15 to 25 percent of portfolio	Establish repeatable migration playbook and validate effort estimates
Wave 2: Core migration	Low to moderate risk	35 to 45 percent of portfolio	Migrate the bulk of the portfolio using the mature playbook
Wave 3: Complex migration	Moderate to high risk	15 to 25 percent of portfolio	Apply strangler fig or parallel run patterns to higher complexity applications
Wave 4: Remediation-dependent migration	High to severe risk	5 to 15 percent of portfolio	Migrate applications requiring prior dependency or architectural remediation

Portfolio share percentages are illustrative defaults; actual distribution depends on the specific risk profile mix present in an organization's portfolio inventory.

9.1 Wave Timing and Overlap

Waves are typically staggered with partial overlap rather than executed in strict sequence, allowing lessons from an earlier wave's later applications to inform a subsequent wave's early applications without requiring the entire prior wave to complete first.

- Typical wave duration for a mid-sized enterprise portfolio, migrating 10 to 30 applications per wave, ranges from 3 to 6 months per wave depending on available engineering capacity.
- A full five-wave portfolio migration for a large enterprise portfolio has been estimated in comparable technology transition planning exercises to span 18 to 36 months from initial pilot through final remediation-dependent migration.

9.2 Communication and Stakeholder Management

Because the roadmap described above sequences applications by technical risk rather than strictly by business unit request order, proactive communication with business stakeholders is essential to maintain organizational confidence in the sequencing rationale, particularly for stakeholders whose applications are assigned to later waves despite organizational pressure for earlier delivery.

- A published, portfolio-wide wave assignment summary, shared with all business stakeholders at the outset of the program, has been associated in comparable enterprise transitions with reduced ad hoc escalation requests relative to programs that communicate wave assignments only individually as each wave begins.
- Regular status communication, typically monthly during active wave execution, covering completed applications, upcoming wave composition, and any material schedule changes, helps sustain stakeholder trust in the risk weighted sequencing approach described in Section 4.2.
- A defined process for business stakeholders to formally request risk reassessment, rather than simply request resequencing, channels legitimate new information, such as a newly identified business criticality change, into the existing scoring model rather than allowing ad hoc exceptions to erode the framework's overall integrity.

10. Effort Estimation Model

Accurate effort estimation is essential for realistic migration wave planning and resource allocation across the roadmap described in Section 9.

Table 11. Illustrative Effort Estimation by Application Size Category

Application Size Category	Typical Screens	Typical Lines of Code	Illustrative Effort (Person-Weeks)
Minimal	Fewer than 10	Under 5,000	2 to 4
Small	10 to 25	5,000 to 20,000	4 to 10
Medium	26 to 60	20,000 to 60,000	10 to 22

Large	61 to 120	60,000 to 150,000	22 to 40
Very large	More than 120	Over 150,000	More than 40

Effort figures assume a moderate overall risk score; applications with elevated custom renderer or dependency compatibility risk scores should apply the adjustment factors in Table 12.

Table 12. Effort Adjustment Factors Based on Weighted Risk Score

Risk Score Range	Effort Adjustment Factor	Rationale
7 to 14 (low overall risk)	0.8 to 1.0 times baseline	Straightforward migration mechanics with minimal unexpected friction
15 to 21 (moderate overall risk)	1.0 to 1.4 times baseline	Some remediation and validation effort beyond baseline mechanics
22 to 28 (high overall risk)	1.4 to 2.0 times baseline	Significant remediation, dependency substitution, or redesign effort
29 to 35 (severe overall risk)	2.0 to 3.0 times baseline	Extensive remediation, likely including partial rewrite or vendor engagement

Adjustment factors are applied multiplicatively to the baseline effort estimate in Table 11 based on the application's weighted risk index computed per Section 5.3.

11. Testing Strategy for Migrated Applications

Testing strategy should be tailored to the migration pattern selected in Section 8 and the existing test coverage risk score described in Section 6.5.

Table 13. Recommended Testing Strategy by Test Type

Test Type	Purpose	Recommended Coverage Target	Typical Tooling Approach
Unit tests	Validate business logic correctness independent of UI framework changes	70 percent or higher for core business logic	Standard .NET unit testing frameworks
UI automation tests	Validate that migrated screens render and behave correctly across platforms	Full coverage of critical user workflows	Cross-platform UI automation frameworks
Visual regression tests	Detect unintended visual differences introduced by the handler architecture change	Full coverage of high-complexity screens identified in Section 6.1	Screenshot comparison tooling
Performance benchmarking	Compare startup time, memory usage, and responsiveness before and after migration	Baseline comparison for all migrated applications	Platform-native profiling tools
Parallel run comparison testing	Compare legacy and migrated application behavior side by side before cutover	Full coverage for parallel run pattern applications per Table 9	Custom comparison harness or manual side-by-side testing

11.1 Test Automation Migration Considerations

Existing automated test suites written against Xamarin.Forms user interface elements frequently require their own migration effort alongside the application code they validate, since UI automation frameworks typically bind to specific control types and accessibility identifiers that may change as custom renderers are replaced with handlers under the new architecture described in Section 3.3. Organizations should treat test suite migration as an explicit line item within the effort estimates described in Section 10 rather than assuming existing test automation will continue to function unmodified against a migrated application.

- Accessibility identifier and automation ID conventions used by existing UI automation suites should be audited early in each application's migration to confirm continued compatibility with the handler based rendering pipeline.
- Unit tests targeting business logic independent of the UI layer generally require little to no modification during migration, since business logic is typically unaffected by the renderer-to-handler architectural change.
- Visual regression test baselines, described in Table 13, should be recaptured after migration rather than compared directly against pre-migration baselines, since minor rendering differences between the two architectures are expected even for functionally correct migrations.

11.2 Regression Test Prioritization Under Time Constraints

Given the effort ranges described in Section 10, migration wave schedules frequently face pressure to compress testing timelines, particularly for applications assigned to later waves under organizational deadline pressure. Rather than uniformly reducing test depth across an entire application, a risk informed prioritization approach concentrates the most rigorous testing on the specific areas flagged as high risk during the assessment described in Section 6, while applying lighter validation to lower risk, unmodified areas of the application.

- Screens or modules identified with elevated user interface complexity scores per Table 5 should receive full visual regression and manual exploratory testing regardless of overall time pressure, given their higher likelihood of introducing subtle rendering defects.
- Business logic areas unaffected by the UI architecture change, and already covered by strong existing unit test coverage per Table 7, can generally rely on regression test suite execution alone without additional manual testing investment.
- A minimum testing floor, below which a migration wave should not proceed to production release regardless of schedule pressure, is commonly defined by steering committees as part of the governance structure described in Section 13.

12. Performance and Platform Behavior Considerations

The shift from the custom renderer architecture to the handler based architecture introduced in .NET MAUI was positioned, at announcement, as reducing per control rendering overhead relative to Xamarin.Forms, though actual performance characteristics require empirical validation on a per application basis given the early, evolving state of the framework at the time of this article's publication.

Table 14. Illustrative Performance Comparison Dimensions (Early Preview Period)

Performance Dimension	Illustrative Xamarin.Forms Baseline	Illustrative Early .NET MAUI Preview Observation	Recommended Validation Approach
Cold start time	1.5 to 3.5 seconds (moderate complexity app)	Directionally comparable to slightly improved in early preview testing	Application-specific measurement across representative devices
Memory footprint (idle)	80 to 150 megabytes (moderate complexity app)	Directionally comparable in early preview testing	Application-specific measurement across representative devices
List and collection scrolling smoothness	Generally smooth with occasional jank on complex custom renderers	Early indications of improved handling for complex custom controls	Manual and automated frame rate testing on target devices
Build and incremental compile time	Baseline for comparison	Early preview tooling still maturing; variable observed results	Track build time trends throughout migration wave execution

Figures reflect early, preliminary observations from the .NET MAUI preview period at the time of this article's publication and should be revalidated against the framework's general availability release before final production decisions.

13. Governance, Risk Register, and Steering

A portfolio scale migration program benefits from a formal governance structure that maintains visibility into risk register status, wave progress, and emerging blockers across the full duration of the multi wave roadmap described in Section 9.

Table 15. Illustrative Migration Program Risk Register

Risk Register Item	Likelihood	Impact	Mitigation Owner	Status
Third-party UI library lacks timely MAUI support	High	High	Architecture team	Monitoring vendor roadmap
Team migration skill gap slows early waves	Moderate	Moderate	Engineering management	Training program underway
Framework preview instability affects schedule	Moderate	High	Program steering committee	Contingency buffer built into roadmap
Business stakeholders request out-of-sequence migration	High	Moderate	Portfolio governance board	Escalation process defined
Shared internal library migration delays dependent applications	Moderate	High	Platform engineering team	Prioritized in Wave 0 and Wave 1

Risk register entries should be reviewed and updated on a recurring cadence, commonly biweekly to monthly, throughout the migration program.

13.1 Program Charter Elements

Establishing a formal program charter at the outset of the migration program, distinct from the ongoing risk register and steering committee cadence described below, helps ensure consistent understanding of program scope, authority, and success criteria across all participating teams before Wave 0 begins.

- A defined program charter typically documents the portfolio scope covered by the migration program, the authority granted to the steering committee to make sequencing decisions per Section 4, and the success criteria against which the program's ultimate outcome will be evaluated.
- Success criteria commonly combine quantitative targets, such as the percentage of the portfolio migrated within a defined timeframe, with qualitative targets, such as maintained or improved production defect rates relative to pre-migration baselines.

13.2 Steering Committee Structure

- A cross functional steering committee typically includes representation from application engineering, platform or architecture teams, quality assurance, and business stakeholders from the most business critical applications in the portfolio.
- Steering committee review cadence of every 2 to 4 weeks is common during active migration wave execution, shifting to monthly during lower activity periods between waves.
- Escalation paths for out of sequence migration requests, noted in Table 15, should be defined in advance to manage business stakeholder expectations against the risk weighted sequencing rationale described in Section 4.2.

13.3 Team Enablement and Training Strategy

Team migration skill readiness, identified as a risk factor in Section 6.6, is best addressed through a structured training strategy that precedes or accompanies early migration waves rather than being left to incidental, on the job learning alone.

Table 15a. Illustrative Training Curriculum by Role

Role	Recommended Training Focus	Illustrative Duration
Mobile application developers	Handler architecture, single project structure, migration tooling hands-on practice	3 to 5 days plus ongoing pilot project experience
Quality assurance engineers	Visual regression and cross-platform test adaptation for the new architecture	2 to 3 days

Engineering managers and team leads	Risk scoring model application and wave planning per Sections 5 and 9	1 to 2 days
Architecture and platform engineers	Shared dependency remediation strategy and vendor engagement per Section 7	2 to 3 days plus ongoing dependency tracking responsibility

- Organizations that complete structured role-based training ahead of Wave 0, rather than during Wave 0 itself, have been associated in comparable technology transitions with shorter pilot wave duration and fewer early defect escapes.
- A dedicated internal knowledge base capturing migration patterns, resolved dependency issues, and reusable code samples, maintained continuously from Wave 0 onward, has been identified as a high value, low cost investment supporting later wave efficiency.

14. Illustrative Simulation Across a Sample Portfolio

To illustrate the framework in application, an illustrative simulation was constructed representing a sample enterprise portfolio of 80 Xamarin applications, applying the risk taxonomy from Section 5 and the wave structure from Section 9.

Table 16. Illustrative Simulation Results: Sample 80-Application Portfolio

Risk Tier	Weighted Score Range	Number of Applications	Assigned Wave	Aggregate Estimated Effort (Person-Weeks)
Minimal to low	7 to 14	22	Wave 0 and Wave 1	150 to 280
Low to moderate	15 to 18	26	Wave 1 and Wave 2	320 to 560
Moderate	19 to 21	18	Wave 2	280 to 450
Moderate to high	22 to 25	9	Wave 3	220 to 380
High to severe	26 to 35	5	Wave 3 and Wave 4	180 to 350

Values reflect an illustrative simulation calibrated to the effort and adjustment ranges in Tables 11 and 12 and should be interpreted directionally rather than as precise figures for any specific organization.

14.1 Interpretation

The illustrative simulation suggests that a typical enterprise portfolio distribution skews toward the lower and moderate risk tiers, with a smaller tail of high and severe risk applications requiring substantially higher per application effort. This distribution supports the wave structure proposed in Section 9, since the large minimal to moderate risk population provides ample opportunity to build organizational migration expertise across Waves 0 through 2 before the smaller, higher risk population is addressed in Waves 3 and 4.

- In the illustrative simulation, applications scoring in the minimal to moderate risk tiers, representing 66 of 80 applications, accounted for approximately 60 to 65 percent of total estimated portfolio migration effort despite representing a larger share of application count, reflecting their lower per application effort.
- The 14 applications in the moderate to high and high to severe tiers, representing under 20 percent of application count, accounted for approximately 35 to 40 percent of total estimated effort, illustrating the effort concentration risk that motivates careful sequencing.

15. Cost and Return on Investment Considerations

Migration investment should be weighed against the risk of continuing to operate on an aging framework lineage, including eventual reduced vendor support prioritization, and the strategic value of aligning with the unified .NET 6 platform direction described in Section 3.2.

Table 17. Illustrative Cost and Benefit Summary for a Full Portfolio Migration Program

Cost or Benefit Category	Illustrative Value (80-Application Portfolio, Full Program)
Total estimated migration engineering effort	1,150 to 2,020 person-weeks (per Table 16)
Dedicated migration program management and governance	150,000 to 350,000 currency units per year across program duration

Training and skill development investment	80,000 to 200,000 currency units (one-time, concentrated in Wave 0)
Avoided future emergency migration cost (compressed timeline scenario)	Estimated 20 to 40 percent cost premium avoided relative to reactive, unplanned migration
Long-term maintenance efficiency from unified single-project structure	10 to 20 percent estimated reduction in per-application maintenance overhead post-migration

Values are presented in currency-neutral illustrative units to remain applicable across currencies and organizational sizes; organizations should substitute local cost figures for planning purposes.

16. Risk Factors and Limitations of the Framework Itself

- The framework's weightings and effort estimates, presented throughout this article as illustrative ranges, are calibrated to early, preliminary .NET MAUI preview experience and should be recalibrated as the framework matures toward general availability and as organizations accumulate their own migration outcome data.
- Risk scores are only as accurate as the underlying inventory data described in Section 5.1; incomplete or outdated dependency and code volume data will produce unreliable prioritization output.
- The framework does not itself resolve third party dependency compatibility gaps; it surfaces and prioritizes them, but resolution still depends on external vendor and open source maintainer roadmaps outside the migrating organization's direct control.
- Organizational change management factors, including business stakeholder pressure for out of sequence migration noted in Table 15, can disrupt risk weighted sequencing regardless of the technical soundness of the underlying prioritization.
- The framework assumes a portfolio large enough to benefit from wave based sequencing; very small portfolios of only a handful of applications may reasonably migrate all applications within a single wave without requiring the full framework.

16.1 Distinguishing Framework Risk from Execution Risk

It is worth distinguishing the inherent limitations of the scoring framework itself, discussed above, from execution risk that arises during actual migration wave delivery regardless of how accurately the initial risk assessment was performed. A well scored, appropriately sequenced application can still encounter unexpected difficulty during execution due to factors not fully captured by the seven factor taxonomy, such as undocumented legacy behavior discovered only once migration work begins, or key personnel turnover affecting institutional knowledge of a specific application's history and edge cases.

- Maintaining a lightweight post migration retrospective process for every completed application, capturing discrepancies between predicted and actual effort, provides the empirical basis needed to recalibrate the framework's weights and effort tables over successive waves, as recommended in Section 17.
- A contingency buffer of 10 to 20 percent added to aggregate wave effort estimates, beyond the per application figures in Tables 11 and 12, is commonly recommended to absorb this category of execution risk without requiring individual application re-scoring for every unexpected finding.

17. Future Research Directions

- Empirical validation of the illustrative effort and risk weighting figures presented in this article against actual migration outcomes as organizations complete portfolio migrations following .NET MAUI's eventual general availability release.
- Automated tooling that extracts portfolio inventory attributes described in Section 5.1 directly from source code repositories at scale, reducing the manual effort currently required to populate the risk scoring model.
- Extension of the risk weighted framework to other concurrent enterprise technology migrations, such as legacy web framework modernization, where similar portfolio scale sequencing challenges arise.
- Longitudinal study of third party dependency ecosystem compatibility maturation over time following a major framework transition, informing more precise timing guidance for dependency-blocked applications identified in Section 7.
- Machine learning based effort estimation refinement that learns from completed migration wave outcomes within an organization to improve the accuracy of the static effort tables presented in Section 10 over successive waves.

17.1 Toward a Reusable Cross-Framework Migration Playbook

A further extension worth pursuing would generalize the risk weighted methodology presented in this article into a reusable playbook applicable beyond this specific Xamarin to .NET MAUI transition, capturing the underlying structural pattern, namely portfolio inventory, multi factor risk taxonomy, weighted scoring, and wave based sequencing, as a framework agnostic methodology. Enterprises face a recurring pattern of large scale framework

and platform transitions over time, and a validated, reusable methodology would reduce the need to reconstruct portfolio migration planning practices from scratch for each subsequent transition.

- Candidate future transitions to which a generalized version of this methodology could plausibly apply include legacy web framework modernization, database platform migration, and cloud infrastructure platform transitions, each of which shares the underlying characteristic of a large, heterogeneous portfolio of affected applications or systems.
- A framework agnostic version would need to abstract the specific risk factors defined in Section 5.2, which are intentionally Xamarin and .NET MAUI specific, into a more general risk factor category structure that individual organizations could populate with transition specific factors.

18. Conclusion

The introduction of .NET MAUI as the architectural successor to Xamarin.Forms presents large enterprises with both an opportunity to align with Microsoft's unified .NET platform direction and a substantial portfolio scale planning challenge. This article has argued that a risk weighted migration framework, combining structured portfolio inventory, a multi factor risk taxonomy, and a weighted scoring model, provides a more defensible and operationally sound basis for sequencing a large scale migration than arbitrary or purely business driven prioritization.

The illustrative simulation presented in Section 14 suggests that most enterprise portfolios will exhibit a skewed risk distribution, with a large population of lower risk applications suitable for early migration waves and a smaller, higher effort tail requiring more deliberate remediation and scheduling. Because .NET MAUI remained in an early, evolving preview state at the time of this article's publication, organizations should treat the specific effort and weighting figures presented here as a structured starting point to be recalibrated against their own migration experience and against the framework's continued maturation toward general availability, rather than as fixed, permanent benchmarks.

More broadly, the risk weighted approach outlined across Sections 5 through 14 reflects a general principle applicable well beyond this specific framework transition: large scale technology migrations are best managed as structured portfolio decisions informed by explicit, revisitable risk assessment, rather than as a collection of independent, ad hoc application level projects. Enterprises that internalize this principle, building the inventory, scoring, and governance habits described in this article as durable organizational capability rather than a one time program artifact, are likely to be better positioned for the next major platform transition, whenever and in whatever form it arrives, than organizations that treat each migration as an isolated, non repeatable event.

Appendix A: Glossary of Key Terms

- Custom renderer. The Xamarin.Forms extensibility mechanism allowing developers to customize how a shared UI control is rendered on a specific platform.
- Handler. The .NET MAUI architectural replacement for custom renderers, intended to reduce per-platform rendering overhead.
- Mac Catalyst. Apple's technology enabling iPad applications to run on macOS, used by .NET MAUI to support desktop macOS targets.
- Mono runtime. The open source .NET runtime implementation historically underlying Xamarin applications prior to .NET unification.
- Parallel run migration. A migration pattern in which the legacy and migrated versions of an application are operated and compared before final cutover.
- Portfolio inventory. A structured record of application attributes used as input to a risk scoring model.
- Risk weighted index. A single composite score computed by combining multiple weighted risk factors for a given application.
- Strangler fig pattern. An incremental migration pattern in which new functionality progressively replaces legacy functionality until the legacy system can be retired.
- Weighted scoring model. A method for combining multiple assessment criteria, each assigned a relative weight, into a single composite score.
- Xamarin.Forms. A cross-platform UI framework allowing shared XAML and C# user interface code to render to native controls across iOS, Android, and Windows.

Appendix B: Consolidated Illustrative Benchmark Data**Table 18. Consolidated Illustrative Benchmark Ranges Referenced Throughout This Article**

Benchmark Item	Illustrative Range
Typical enterprise Xamarin portfolio size	20 to 300 or more applications
Applications with significant custom renderer usage	30 to 60 percent of applications with nontrivial UI
Minimal application migration effort	2 to 4 person-weeks
Very large application migration effort	Over 40 person-weeks
Typical migration wave duration	3 to 6 months
Full five-wave program duration (large portfolio)	18 to 36 months
Shared dependencies driving majority of compatibility risk	Often fewer than 20 distinct packages
Steering committee review cadence (active wave execution)	Every 2 to 4 weeks

Ranges are illustrative and drawn from patterns reported across comparable enterprise application modernization programs discussed in this article.

Appendix C: Risk Scoring Worksheet Template

The following worksheet structure can be reused directly by migration planning teams to score an individual application against the seven factor taxonomy defined in Section 5.2.

Table 19. Risk Scoring Worksheet Template for a Single Application

Risk Factor	Weight	Raw Score (1-5)	Weighted Contribution
User interface complexity	15 percent	[to be assessed]	[weight x score]
Custom renderer dependency	20 percent	[to be assessed]	[weight x score]
Third-party library compatibility	20 percent	[to be assessed]	[weight x score]
Platform-specific code exposure	15 percent	[to be assessed]	[weight x score]
Automated test coverage	15 percent	[to be assessed]	[weight x score]
Team migration skill readiness	10 percent	[to be assessed]	[weight x score]
Release cadence and change freeze exposure	5 percent	[to be assessed]	[weight x score]

Sum the weighted contribution column to obtain the application's composite weighted risk index for comparison against the scale in Table 4 and wave assignment in Table 10.

Appendix D: Sample Stakeholder Communication Plan Template

The following template, referenced from the stakeholder management guidance in Section 9.2, can be adapted directly by migration program leads to structure recurring communication throughout the multi wave roadmap.

Table 19a. Sample Stakeholder Communication Plan Template

Audience	Communication Type	Frequency	Primary Content
Executive sponsors	Program status briefing	Monthly	Overall roadmap progress, budget status, major risks from Table 15
Business unit stakeholders (all)	Portfolio-wide wave assignment summary	One-time at program launch, refreshed after major replanning	Full wave assignment list with sequencing rationale
Business unit stakeholders (active wave)	Wave kickoff and status update	At wave start, then monthly during wave execution	Application-specific timeline, testing

			approach, and go-live plan
Engineering teams (all)	Migration playbook and lessons learned update	After each completed wave	Updated patterns, resolved dependency issues, and revised effort estimates
Steering committee	Detailed risk register and schedule review	Every 2 to 4 weeks during active execution	Full risk register per Table 15, schedule variance, escalation items

Frequencies are illustrative defaults; programs operating a larger portfolio or facing elevated organizational scrutiny may warrant more frequent communication for select audiences.

Consistent use of a defined communication plan, rather than ad hoc updates driven by whichever stakeholder raises a concern most recently, reinforces the perceived fairness and rigor of the risk weighted sequencing approach described throughout this article, which in turn supports the change management objectives discussed in Section 9.2.

Appendix E: Program Readiness Checklist

- Confirm complete portfolio inventory data has been collected for all applications in scope, per the attributes defined in Table 2.
- Confirm risk factor weights in Table 3 have been reviewed and, if necessary, adjusted to reflect organizational risk tolerance before scoring begins.
- Confirm a shared dependency compatibility tracking process, described in Section 7.2, is established before Wave 0 begins.
- Confirm steering committee membership and review cadence, described in Section 13.1, are formally established.
- Confirm training and skill development investment, described in Section 15, is planned to precede or accompany Wave 0 and Wave 1 execution.
- Confirm a process exists for periodically refreshing risk scores as described in Section 4.2, given the evolving state of the .NET MAUI ecosystem at the time of this article's publication.

REFERENCES

- 1) Cunningham, W. The WyCash Portfolio Management System. OOPSLA Experience Report, 1992.
- 2) Forsgren, N., Humble, J., and Kim, G. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution Press, 2018.
- 3) Fowler, M. StranglerFigApplication. Martin Fowler's Bliki, 2004.
- 4) Fowler, M. TechnicalDebt. Martin Fowler's Bliki, 2003.
- 5) Fowler, M., and Sato, D. FeatureToggle. Martin Fowler's Bliki, 2010.
- 6) Hermes, D.B. Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals. Apress, 2015.
- 7) International Organization for Standardization. ISO 31000: Risk Management Guidelines. ISO, 2018.
- 8) Kim, G., Humble, J., Debois, P., and Willis, J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, 2016.
- 9) Microsoft Corporation. .NET 6 Preview Release Notes. Microsoft Docs, 2021.
- 10) Microsoft Corporation. .NET Multi-platform App UI Announcement. Microsoft Build Developer Conference, May 2021.
- 11) Microsoft Corporation. Xamarin.Forms Architecture and Application Fundamentals. Microsoft Docs, 2019.
- 12) Microsoft Corporation. Xamarin.Forms Documentation: Custom Renderers. Microsoft Docs, 2020.
- 13) Newman, S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.
- 14) Newman, S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, 2019.
- 15) Petzold, C. Creating Mobile Apps with Xamarin.Forms. Microsoft Press, 2016.
- 16) Project Management Institute. A Guide to the Project Management Body of Knowledge (PMBOK Guide). 6th Edition. PMI, 2017.
- 17) Richardson, C. Microservices Patterns: With Examples in Java. Manning Publications, 2018.