Ca' Foscari University of Venice

Master Degree
in Computer Science and Information Technology

Final Thesis

Orchestration Made Simple: A Practical Analysis of Aspire in a Cloud-Native Environment

Supervisor
Ch. Prof. Pietro Ferrara

Graduand
Elia Bertapelle
Matriculation Number 881359

Academic Year
2024 / 2025

Abstract

The transition from monolithic architectures to microservices has revolutionized software scalability; however, this shift has introduced significant operational complexities and increased the cognitive load on development teams. This thesis explores Aspire, an opinionated, cloud-native stack designed to streamline the development, integration, and orchestration of distributed systems. The work evaluates the framework's ability to abstract infrastructure concerns through its high-level orchestration model, standardized component system, and native instrumentation. To assess its practical applicability, a modular business operations platform (CRM) was designed and implemented for I-Tech S.r.l., transitioning from a legacy PHP monolith to a modern distributed environment.

The study specifically analyzes how the Aspire dashboard enhances the debugging process, the efficiency gains provided by automated service discovery, and the integration of OpenTelemetry standards for distributed tracing. Experimental results demonstrate that Aspire effectively mitigates the "Inner Loop" frictions inherent in microservices development by providing a unified environment for both local and cloud-based contexts. By automating boilerplate infrastructure configuration and providing "zero-configuration" telemetry, the framework allows developers to focus on core business logic. Ultimately, this thesis proves that Aspire is a critical solution for reducing architectural complexity and lowering the barrier to entry for cloud-native adoption in small-to-medium enterprises.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Pietro Ferrara, for his invaluable guidance and constant support, especially during the moments when I struggled to find the right direction. His insight was fundamental in keeping this research on track.

A special thanks goes to everyone at I-Tech S.r.l. Having the opportunity to conduct this thesis within the company where I work allowed me to grow in a professional environment where I felt truly supported. The ability to focus on improving our current technologies while having direct access to the company's resources and expertise was crucial to the success of this project. I want to thank my colleagues for the immense patience and support they have shown me throughout this entire process.

I am also deeply grateful to my family, my parents and my sisters. Your unwavering belief in my abilities and your constant encouragement provided the foundation I needed to reach this milestone. I truly could not have done this without you.

A big thank you to my girlfriend, Vera, who has been my rock throughout this journey. Your love, understanding, and support have been invaluable, especially during the most challenging moments. You have been by my side every step of the way, and I am incredibly grateful for your presence in my life.

Finally, a heartfelt thank you to all my friends who have stood by me. To my childhood friends, who have known me since the beginning; to my family friends, whose own journeys inspired me; to my friends from ACR, for the years of shared growth and experiences; and to my university colleagues, who shared the unique challenges of this academic path. Thank you all for the necessary distractions, the constant encouragement, and for helping me carry the load during the most demanding phases of this journey.

Contents

1 Introduction **1**

1.1 Context 1

1.2 Problem Statement 3

1.3 Objective 3

1.3.1 Questions 4

1.4 Scope / Delimitations 4

1.5 Structure of the Thesis 5

2 Background **6**

2.1 Software Architectures 6

2.1.1 The Monolithic Paradigm 7

2.1.2 Distributed and Microservices Architectures 8

2.2 The Cloud-Native Paradigm 9

2.2.1 The Development Lifecycle: Inner and Outer Loop 10

2.2.2 Orchestration 11

2.2.3 Service Discovery 12

2.3 Cross-Cutting Concerns 14

2.3.1 Infrastructure as Code (IaC) 14

2.3.2 Observability and OpenTelemetry 15

2.3.3 Identity and Security 16

2.4 The Development Ecosystem 18

2.4.1 .NET Technologies 18

2.4.2 Kestrel 18

2.4.3 Blazor 19

2.4.4 Entity Framework (EF) 19

2.4.5 Swashbuckle 19

2.4.6 TabBlazor and Tabler 20

2.4.7 Polly 20

2.4.8 Visual Studio 21

2.5 Supporting Infrastructure . . . 22
2.5.1 PostgreSQL . . . 22
2.5.2 RabbitMQ . . . 22
2.5.3 Docker . . . 23
2.5.4 Kubernetes (K8s) . . . 25

3 Aspire **27**
3.1 Introduction . . . 27
3.2 Evolution and Versioning . . . 28
3.3 AppHost and The Code-First Paradigm . . . 28
3.3.1 Architectural Analysis of the AppHost . . . 30
3.3.2 Resource Orchestration and Lifecycle . . . 30
3.3.3 Service Discovery and Network Abstraction . . . 31
3.3.4 Health Monitoring and Resource Readiness . . . 32
3.4 Service Defaults and Standardized Governance . . . 33
3.4.1 Functional Components of Service Defaults . . . 33
3.4.2 Extensibility and Customization . . . 34
3.5 Observability and the Distributed Dashboard . . . 34
3.6 Networking and Service Discovery . . . 35
3.6.1 Endpoint Management and Proxies . . . 36
3.6.2 Container Networking . . . 36
3.6.3 Configuration Sources: Launch Profiles and Kestrel . . . 36
3.6.4 Container Port Mapping . . . 37
3.6.5 Service Discovery Mechanics . . . 37
3.7 Aspire Components and Integrations . . . 37
3.7.1 Polyglot and Multi-language Support . . . 38
3.8 Deployment Strategies . . . 39
3.8.1 The Specification-Execution Split . . . 39
3.8.2 Hosting Integrations and Compute Environments . . . 39
3.8.3 Parameterization and Security . . . 40
3.8.4 Transitioning from Legacy Manifests . . . 40

4 Case Study: I-Tech S.r.l. **41**
4.1 Introduction . . . 41
4.2 State of the Art . . . 42
4.2.1 Architecture and Deployment . . . 42
4.2.2 Monolithic Limitations and Performance Constraints . . . 43
4.3 Project Goals . . . 45

4.4 Design Requirements 46
4.4.1 Functional Requirements 46
4.4.2 Non-Functional Requirements 47

5 Implementation **49**
5.1 Technical Infrastructure and Tooling 49
5.2 Project Genesis and Initial Stalling 49
5.3 Aspire integration 51
5.3.1 The Unified Development Stack 52
5.4 Orchestration with the AppHost 53
5.5 Infrastructure and Service Binding 56
5.6 Security Implementation and Homogeneity 58
5.7 Deployment Pipeline: From AppHost to Cloud 59
5.7.1 Implementation Strategy: On-Premise Docker Testing . . . 59
5.7.2 The Path to Production: Azure Kubernetes Service (AKS) . 60
5.7.3 CI/CD Integration and Extensibility 60
5.8 Summary of the Implementation 60

6 Evaluation and Discussion **62**
6.1 Observability in action 62
6.2 Maintainability and System Evolution 63
6.3 Developer Experience (DX) 65
6.4 Resilience Testing and Fault Tolerance 67
6.5 Consistency and Standardization 67
6.5.1 Standardized Cross-Cutting Concerns 68
6.5.2 Security Homogeneity and Contract Safety 68
6.6 Cost-Benefit Analysis 69
6.7 Strategic Advantages of the Open-Source Ecosystem 70
6.8 Discussion of Initial Questions 71

7 Conclusion **73**
7.1 Aspire Strengths 73
7.2 Limitations 74
7.3 Future Work 75
7.3.1 Aspire Development 75
7.3.2 Evolution of the I-Tech CRM 75
7.4 Final Remarks 75

Chapter 1

Introduction

For over a decade, the landscape of software engineering has been steadily transitioning from monolithic architectures toward distributed, cloud-native systems. While this shift driven by the need for massive scalability and rapid deployment cycles is now well-established, it has introduced a persistent layer of operational complexity that often compromises developer productivity. This thesis explores the implementation of Aspire, a modern, opinionated stack designed to manage the orchestration and observability of these distributed systems. Through the lens of a real-world modernization project, this work analyzes how Aspire mitigates the long-standing challenges of service discovery and local development within the .NET ecosystem.

1.1 Context

This thesis is conducted in collaboration with I-Tech S.r.l., an Italian software house specializing in customized ERP integrations and technical support. While the company has historically relied on monolithic, on-premises applications, the evolving digital landscape has necessitated a shift toward cloud-native architectures. This transition is driven by the need for the agility and scalability required to meet modern business demands.

The first step in this transformation was identified as the modernization of I-Tech's legacy Customer Relationship Management (CRM) system. Since 2018, this platform has served as the primary engine for support ticket management, tracking customer interactions, and coordinating technical interventions. As the central repository for customer data and service history, the CRM is a mission-critical tool that directly impacts the company's ability to maintain high service-level agreements (SLAs).

Historically, I-Tech has relied on a CRM authored in PHP and maintained by a third-party vendor. This software was built on a traditional monolithic architecture and deployed on-premises. While this setup was reliable within a controlled, internal environment, it has increasingly struggled to align with evolving organizational needs. The rigid structure of the PHP monolith created significant technical debt, rendering the implementation of critical new business features nearly impossible within the existing codebase.

I-Tech identified that the current CRM system suffered from significant performance degradation under concurrent loads. Specifically, the planned addition of a Customer Portal (allowing clients to open and track tickets directly) and real-time synchronization with the Mago4 ERP threatened to push the monolith beyond its operational threshold. Opening the system to external traffic would have likely resulted in cascading failures and persistent service unavailability, as the single-threaded nature of critical legacy components created bottlenecks that could not be resolved without a complete architectural decoupling.

In late 2022, I-Tech initiated a greenfield rewrite of the CRM as a distributed system. However, the early stages of migration exposed several systemic "frictions" inherent in the shift to cloud-native development. A primary obstacle was the degradation of the "Inner Loop" development cycle; while monolithic architectures facilitate a unified execution environment allowing developers to launch and debug the entire system in a single process, a microservices landscape introduces fragmented workflows. In this new paradigm, managing the lifecycle of disparate containers, databases, and APIs becomes a labor-intensive manual task. This operational complexity is further exacerbated by observability gaps, where service-to-service communication necessitates extensive manual instrumentation to yield actionable logs, metrics, and distributed traces.

Furthermore, the team encountered significant challenges regarding service discovery and configuration, where the manual management of connection strings often led to "configuration drift" across different environments. These technical obstacles contributed to a high cognitive load, as the steep learning curve of orchestrators like Kubernetes often forced developers to focus more on boilerplate YAML manifests and network configurations than on core business logic. While the team evaluated industry-standard solutions like Docker and Kubernetes, the limited timeframe and the complexity of these tools initially led to the project's postponement, highlighting the need for a more streamlined orchestration platform.

1.2 Problem Statement

The core problem addressed by this thesis is the high "infrastructure tax" associated with transitioning from legacy monolithic systems to cloud-native distributed architectures. While microservices offer superior scalability and fault tolerance, the transition introduces a significant operational overhead that often exceeds the capacity of small engineering teams.

This challenge is primarily manifested in the degradation of the "Inner Loop" development cycle. In a monolithic environment, developers benefit from a unified execution context; however, in a distributed landscape, the local development, orchestration, and debugging phases become fragmented, thereby creating a barrier that small teams cannot overcome. Without a streamlined way to manage service interdependencies and observability, modernization efforts frequently result in decreased developer velocity and increased technical debt, leaving smaller organizations trapped in aging monolithic structures.

1.3 Objective

The primary objective of this thesis is to critically evaluate the efficacy of Aspire as a transformative tool for orchestrating distributed systems within a real-world modernization context. Rather than merely documenting a migration, this thesis seeks to determine the extent to which an "opinionated" stack can mitigate the inherent complexities of cloud-native development for small-to-medium enterprise (SME) teams.

Central to this investigation is the assessment of Aspire's "dashboard-first" philosophy and its impact on the developer experience. The work focuses on three key pillars: the reduction of cognitive load through automated local orchestration, the streamlining of service discovery and configuration management, and the immediate accessibility of system-wide observability through integrated telemetry. By implementing these features within the redesign of the I-Tech CRM system, this work aims to provide a practical roadmap for transitioning from rigid monolithic architectures to resilient distributed systems without the prohibitive overhead typically associated with cloud-native tooling.

Furthermore, this thesis intends to provide a comparative reflection on traditional orchestration methods versus the Aspire model. Through the lens of this case study, we evaluate how Aspire influences developer velocity specifically addressing the "Inner Loop" friction and whether its abstraction of infrastructure concerns al-

lows developers to maintain a higher focus on business logic. Ultimately, this thesis aims to validate whether Aspire represents a viable middle ground for teams that require the scalability of microservices but possess limited resources for managing complex infrastructure orchestrators like Kubernetes.

1.3.1 Questions

To evaluate the efficacy of Aspire in addressing these challenges, this thesis seeks to answer the following research questions:

- Q1: To what extent does Aspire reduce the setup time and operational complexity for developers transitioning from monolithic to microservices architectures?
- Q2: How effectively does a "dashboard-first" observability approach improve the detectability and resolution of failures in a distributed system compared to manual instrumentation?
- Q3: Can high-level abstractions sufficiently bridge the skills gap for a small development team, allowing them to deploy production-ready distributed systems without deep expertise in complex orchestrators like Kubernetes?

1.4 Scope / Delimitations

This thesis focuses specifically on the development and local orchestration phase of the software lifecycle using Aspire. While the transition to a distributed system involves many facets, the following boundaries apply to this study:

Technological Focus
Even if Aspire supports multiple technologies and integrations, this thesis concentrates on a limited subset: ASP.NET Core for the backend, PostgreSQL for data storage, RabbitMQ for messaging, and Blazor for the front-end interface.

Organizational Context
The evaluation is tailored to the needs of a Small-to-Medium Enterprise (SME) and may not reflect the infrastructure requirements of massive-scale global enterprises.

Exclusions
This thesis does not provide an exhaustive security audit of the new system, nor does it perform a detailed financial analysis of cloud hosting costs. The primary metric for success is the improvement in developer velocity and system observability rather than raw runtime performance benchmarks.
Considering the limited timeframe, many implementation details and optimizations are deferred to future work, focusing instead on establishing a functional baseline for comparison.

1.5 Structure of the Thesis

This thesis is organized into seven chapters to provide a comprehensive analysis of the modernization process. Following this introduction, Chapter 2 (Background) establishes the theoretical framework, detailing cloud-native pillars and the .NET ecosystem. Chapter 3 (Aspire) provides a deep dive into the platform's orchestration model and its "dashboard-first" approach.

The practical application begins in Chapter 4 (Case Study), which outlines the functional requirements and the decomposition of the legacy CRM. Chapter 5 (Implementation) serves as the case study, documenting the integration of RabbitMQ, PostgreSQL, and Blazor using Aspire. The final stages of this work, found in Chapter 6 (Results and Evaluation) and Chapter 7 (Conclusions), analyze the impact on developer velocity and summarize the findings and future work directions.

Chapter 2

Background

Starting from the ideas introduced in the previous chapter, this section provides the necessary theoretical background to understand the concepts and technologies involved in this thesis. This foundation is essential to grasp the motivation behind the development of Aspire and how this platform aims to address the inherent challenges developers face when working with microservices and cloud-native architectures.

2.1 Software Architectures

We can define a software architecture as the structural backbone of a software system and, as noted by Richards and Ford [35], it includes an architectural style along with the features it must support, the logical components, and the decisions that support the design.

The choice of the architectural style is crucial as it directly impacts the system's scalability, maintainability, and overall performance. For this reason, understanding the difference between various architectural styles is fundamental when designing modern applications. A typical software system is composed of three main layers: the presentation layer (user interface), the logic layer (application functionality), and the data layer (data storage and retrieval). How these layers are organized and interact defines the architectural style and they can be structured in various ways.

In the context of this thesis, we focus on two primary architectural styles: monolithic and distributed (with an emphasis on microservices) and their implications for cloud-native development.

2.1.1 The Monolithic Paradigm

The monolithic architecture represents the most traditional paradigm in software engineering, characterized by a design where all functional components reside within a single codebase and execute as a unified process. Historically, this approach has been favored for its simplicity; however, as systems evolve in scale and complexity, the monolithic model often encounters significant structural challenges. This paradigm manifests in various patterns, ranging from the Layered (n-tier) Architecture, which attempts to organize concerns into horizontal layers, to the Microkernel Architecture, which extends a minimal core through specialized plug-ins. Conversely, many long-lived monoliths eventually devolve into a "Big Ball of Mud", a recognized anti-pattern defined by high coupling and a lack of formal boundaries, which renders maintenance and evolution increasingly difficult.

Despite the modern trend toward distribution, the monolithic style offers distinct advantages, particularly regarding development velocity and system performance. As noted by prof. Ferrara during the Software Architecture course [5], the primary benefit lies in the ease of development; a single codebase allows developers to navigate, refactor, and debug the entire application without the complexities of network boundaries. Furthermore, since all components share the same memory space, the system benefits from low latency and high performance, avoiding the serialization overhead inherent in distributed communication. For small teams or applications with limited complexity, the operational simplicity of managing a single deployment unit often results in a lower total cost of ownership.

However, the monolith's centralized nature eventually becomes a liability during expansion. The most critical limitation is in scalability; because the application is a single unit, it must be scaled vertically (by adding more resources to a single server) or replicated in its entirety, which leads to inefficient resource utilization. This structural rigidity is mirrored in the system's fault tolerance; a memory leak or a fatal error in a secondary component can compromise the entire process, leading to total system failure. Furthermore, the monolithic model often struggles with modern delivery demands. Any modification, regardless of how minor, necessitates a full redeployment of the application, which lengthens deployment cycles and increases the risk of downtime. Over time, as the codebase grows, it becomes a "legacy" system that is difficult to adapt to new technologies, as the entire stack is locked into the original architectural choices. While the monolith remains an excellent choice for initial prototypes and small-scale deployments, the transition to a distributed architecture becomes an inevitable necessity when an application must scale, adapt, and remain resilient in a competitive cloud environment.

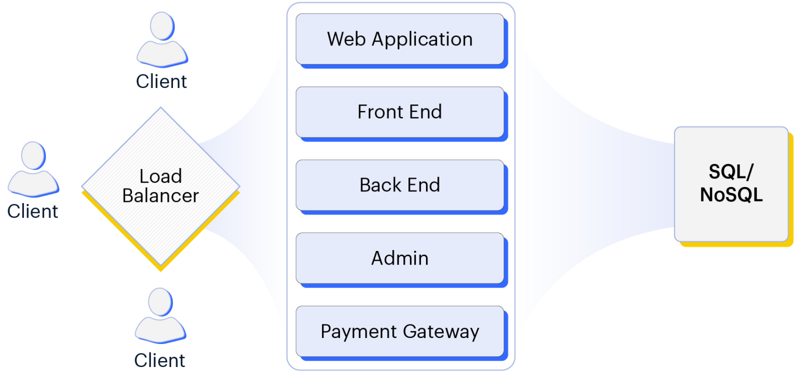


Figure 2.1: Monolithic Architecture Diagram [42]

2.1.2 Distributed and Microservices Architectures

When the inherent limitations of monolithic architectures (such as deployment bottlenecks and scaling inefficiencies) become apparent, organizations often pivot toward distributed architectures. This shift has gained significant traction alongside the maturation of cloud computing, which provides the necessary infrastructure for scalable, resilient applications. The fundamental premise of distributed systems is the decomposition of an application into smaller, autonomous services. While several patterns exist within this paradigm, they differ primarily in their level of decoupling. Service-Based Architectures allow for independent deployment but often maintain a shared database, whereas Event-Driven Architectures (EDA) focus on asynchronous communication through event brokers to handle real-time state changes.

The most prominent of these patterns is the Microservices Architecture, as defined by industry literature, microservices are independently deployable units modeled around specific business domains rather than technical layers. Unlike traditional models, this approach emphasizes strict data ownership, where each service encapsulates its own data schema and retrieval logic. This encapsulation prevents the tight coupling often seen in shared-database environments and allows for "polyglot persistence" the ability to use different storage technologies tailored to the specific needs of each service.

The transition to microservices offers substantial advantages, particularly regarding system resilience and organizational agility. By enabling horizontal scaling, individual services can be expanded based on specific demand without the need

to replicate the entire application. Furthermore, the isolation of failures ensures that a crash in a non-critical service does not necessarily lead to a total system outage. However, these benefits are inextricably linked to a surge in operational complexity. Shifting from in-memory function calls to network-based communication introduces challenges such as network latency, serialization overhead, and the loss of traditional ACID transactions.

Consequently, microservices demand a higher level of operational maturity. Managing a distributed environment requires sophisticated tooling for service discovery, centralized logging, and distributed tracing to maintain visibility across service boundaries. While this architecture provides the flexibility and technology diversity necessary for large-scale, high-availability systems, it also imposes a significant "infrastructure tax" that must be managed through automation and opinionated orchestration frameworks.

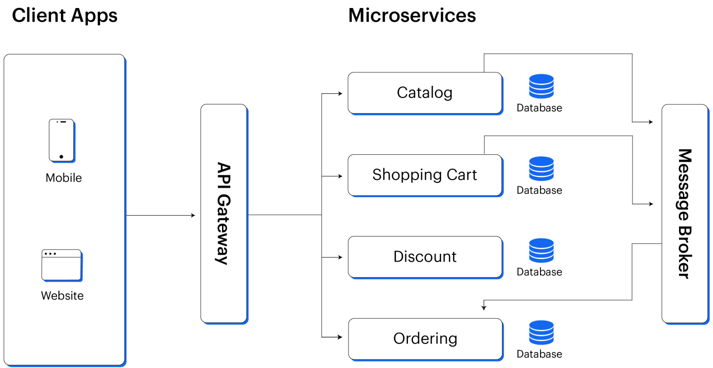


Figure 2.2: Microservices Architecture Diagram [43]

2.2 The Cloud-Native Paradigm

The term "cloud-native" describes a specialized approach to designing, building, and running applications that fully exploit the advantages of the cloud computing delivery model. It represents a fundamental shift in software engineering; rather than simply migrating legacy systems to remote servers (a process often referred to as "lift and shift"), cloud-native development involves re-architecting applications to thrive within dynamic, virtualized environments. These systems are characterized by their ability to remain scalable, resilient, and manageable, typically achieving these traits through the synergy of microservices architectures, containerization, and automated orchestration.

A truly cloud-native application is defined by several core architectural principles that distinguish it from traditional on-premises software. Central to this is

the concept of statelessness, where services are designed to operate without storing client session data locally. This allows individual instances to be destroyed, replaced, or replicated across a cluster without disrupting the user experience. This design directly supports elastic scalability, enabling the system to utilize horizontal scaling (adding or removing service instances in real-time) to meet fluctuating workload demands efficiently.

Furthermore, cloud-native systems prioritize resilience through fault-tolerant patterns, such as circuit breakers and self-healing mechanisms, which allow the application to recover from infrastructure failures automatically. However, as the system becomes more distributed and dynamic, the need for observability becomes paramount. Unlike monolithic logging, cloud-native observability requires integrated telemetry (comprising metrics, distributed traces, and centralized logs) to provide deep insights into service-to-service communication and system health. By adhering to these principles, organizations can transition from rigid, fragile release cycles to a continuous delivery model that supports the agility required by the modern digital landscape.

2.2.1 The Development Lifecycle: Inner and Outer Loop

The modern Cloud-Native Development Lifecycle is characterized by two distinct cycles that address different aspects of software development and deployment: The Inner Loop and The Outer Loop. The efficiency of a cloud-native organization is fundamentally determined by the latency within these two loops.

The Inner Loop: The Micro-Iteration of the Individual

The Inner Loop represents the iterative process performed by an individual developer prior to sharing code with a version control system. In an academic context, this is defined as the "experimental phase", where the primary objective is to rapidly validate local changes. This cycle typically encompasses the "code-build-run-test" sequence.

In traditional architectures, the Inner Loop was relatively low-friction; however, the cloud-native transition introduces significant complexity. Because the target production environment (e.g., Kubernetes) is fundamentally different from a local workstation, developers face a "Cognitive Load" penalty. To maintain the "flow state" necessary for deep engineering, the Inner Loop must minimize the duration between a code edit and the observation of its results. Key challenges here include the latency introduced by container image builds and the difficulty of mocking distributed cloud dependencies locally.

The Outer Loop: The Macro-Iteration of the System

The Outer Loop starts once code is committed to a shared repository, triggering the automated processes of the organization. This is the "validation and delivery phase", where the focus shifts from individual productivity to collective quality, security, and stability.

The Outer Loop includes formal Continuous Integration (CI), automated security gatekeeping (DevSecOps), peer code reviews, and Continuous Deployment (CD) to staging and production environments. While the Inner Loop is optimized for velocity, the Outer Loop is optimized for rigor. In a cloud-native paradigm, the Outer Loop is increasingly characterized by GitOps principles, where the desired state of the infrastructure is managed declaratively alongside the application code.

The Convergence: Environment Parity and the "Friction Point"

The intersection of these two loops is the most critical juncture in the cloud-native lifecycle. A primary research problem in this field is the "Parity Gap" the technical divergence between the developer's local Inner Loop and the production-grade Outer Loop.

When the Inner Loop does not accurately simulate the constraints of the Outer Loop (such as service meshes, ingress controllers, or security policies), "logic leakage" occurs. This results in bugs that are only discoverable late in the Outer Loop, leading to costly context-switching and rework. Consequently, modern cloud-native research focuses on "Loop Synchronization", utilizing tools such as Telepresence or Skaffold to project Outer Loop infrastructure into the Inner Loop environment, thereby achieving high-fidelity feedback earlier in the lifecycle.

2.2.2 Orchestration

The automated arrangement, coordination, and management of complex computer systems, middleware, and services is formally recognized as orchestration. Within the paradigm of cloud-native architectures and microservices, orchestration specifically refers to the comprehensive lifecycle management of containers or virtualized units. This process ensures that the desired state of a distributed system, as defined by the architect, is continuously maintained by the underlying infrastructure. Unlike simple automation, which focuses on the execution of discrete, repetitive tasks, orchestration involves the sophisticated alignment of multiple automated processes to manage interdependencies, networking, and workflows across heterogeneous environments.

Building upon this foundational coordination, the primary responsibility of an orchestration engine is the intelligent abstraction of hardware resources. By treating a cluster of physical or virtual machines as a single, unified pool of computational power, the orchestrator simplifies the deployment process through advanced scheduling and resource management. It evaluates the specific requirements of a service such as CPU cycles, RAM allocation, and storage persistence and determines the optimal placement for that service based on real-time telemetry and predefined affinity rules. This level of abstraction is critical for maintaining operational stability, as it allows the system to implement self-healing mechanisms. When the orchestrator detects a node failure or a service crash, it automatically re-provisions the affected components to restore the system to its intended operational baseline without manual intervention.

This management logic is fundamentally rooted in the shift from imperative to declarative configuration models. In an imperative model, an operator must define the specific sequence of commands required to achieve a result; however, modern orchestration utilizes a declarative approach where the operator defines only the final "desired state". The orchestrator then employs a continuous reconciliation loop, a core logic cycle that perpetually compares the current state of the infrastructure against the target state. If a discrepancy is found such as a missing service instance or a localized traffic bottleneck the orchestrator executes the necessary actions to bridge the gap. This mechanism not only facilitates horizontal scaling, where instances are dynamically added or removed in response to fluctuating workloads, but also enables complex deployment strategies like canary releases and automated rollbacks, which minimize the risks associated with software updates.

Ultimately, orchestration provides the necessary decoupling of the application layer from the physical infrastructure, which is a prerequisite for modern scalability and portability. By standardizing how services are deployed and managed, orchestration frameworks allow for consistent application behavior across diverse environments, including private data centers and public cloud providers. This abstraction layer ensures that as the complexity of the microservices ecosystem grows, the operational overhead remains manageable, allowing the system to scale in both size and capability while maintaining high availability and performance standards.

2.2.3 Service Discovery

Service discovery serves as a fundamental mechanism for achieving network location transparency within distributed systems. It enables disparate services to locate

and communicate with one another without requiring manual configuration for IP addresses or hostnames. In modern, cloud-native environments characterized by ephemeral infrastructure, service instances are frequently provisioned, terminated, or rescheduled across different physical nodes. This dynamic nature renders static networking configurations obsolete. Service discovery addresses this challenge by providing a systematic method for automatically detecting and tracking the network locations of services in real-time as they oscillate within the cluster.

Central to this architecture is the Service Registry, a highly available database that acts as a definitive source of truth for the entire ecosystem. The lifecycle of a service in this context begins with the registration process, wherein an instance broadcasts its network coordinates a combination of IP address and port to the registry upon initialization. To ensure the integrity of the registry, many systems implement a "heartbeat" mechanism, where the registry periodically verifies the health of the registered instances. If an instance becomes unresponsive or undergoes a graceful shutdown, it is deregistered, thereby preventing consumers from attempting to route traffic to non-functional or non-existent endpoints. This automated management of connectivity is what facilitates the seamless, high-velocity communication required in a microservices architecture.

The architectural implementation of this discovery process generally bifurcates into two distinct patterns: Client-Side Discovery and Server-Side Discovery. In the client-side pattern, the responsibility for service location is delegated to the service consumer. The client queries the service registry directly, receives a list of available instances, and utilizes a local load-balancing algorithm to select a specific target. While this approach reduces the number of network hops and provides the client with more granular control over load balancing, it increases the complexity of the client-side code and requires the developer to implement discovery logic across multiple programming languages or frameworks.

On the other hand, server-side discovery abstracts this complexity by introducing an intermediary component, such as a load balancer or an API gateway. In this model, the client issues a request to the intermediary, which then queries the service registry and routes the traffic to an appropriate instance on the client's behalf. This pattern simplifies the client-side implementation, as the consumer remains unaware of the underlying discovery mechanics. However, it introduces an additional network hop and may create a centralized point of failure or a performance bottleneck if the intermediary is not properly scaled. Ultimately, the choice between these patterns depends on the specific requirements for system latency, developer overhead, and the desired degree of infrastructure abstraction.

2.3 Cross-Cutting Concerns

Cross-cutting concerns encompass functionalities that apply across the entire application rather than specific business domains. Addressing these consistently is vital for maintaining architectural integrity and operational stability.

2.3.1 Infrastructure as Code (IaC)

Infrastructure as Code (IaC) marks an important change in system administration. It shifts from manual, error-prone configurations to a model where infrastructure is managed with machine-readable definition files. By treating environment provisioning like a software engineering task, IaC allows teams to use standard development practices such as version control, continuous integration, and automated testing for the hardware and network layers. This method ensures that environments are reproducible, consistent, and documented within the codebase, effectively eliminating the "configuration drift" that often occurs in manually managed systems.

Traditionally, IaC has used a declarative approach, relying on static data formats like YAML or JSON to describe a desired final state. While effective, these formats often lack the abstraction and logic found in general-purpose programming languages. This limitation has led to the rise of "Code-First" approaches, which let developers define their application's infrastructure and dependencies using familiar languages like C#. This change reduces the cognitive load on developers by removing the need to switch between application logic and specialized configuration languages. Later in this thesis, we will explore how Aspire makes use of this approach to provide a unified development model. By defining the application's components, such as databases, caches, and inter-service dependencies, directly in C#, Aspire can automatically generate the necessary deployment artifacts, like Kubernetes manifests, ensuring that the infrastructure stays aligned with the application's changing requirements.

Fluent API and Declarative Orchestration

Fluent API is a software engineering design pattern aimed at increasing code readability by creating a "Domain-Specific Language" (DSL) within the host programming language. Coined by Eric Evans and Martin Fowler, the pattern relies heavily on Method Chaining, where each method call returns a reference to an object (typically the same instance) allowing subsequent configurations to be appended in a single, continuous statement.

In orchestration frameworks, this pattern transforms Imperative Configuration

into a declarative narrative. Unlike static configuration formats like YAML or JSON, a Fluent API provides:

- **Type Safety**: Leveraging the compiler to catch configuration errors (e.g., assigning a string to a port number) at compile-time rather than deployment-time.
- **IntelliSense Discoverability**: Reducing the cognitive load on the developer by surfacing only the valid subsequent configurations available for a specific resource type.

```
// 1. Standard Imperative Way
// Requires multiple references to the variable and manual
   property setting.
  EmailMessage msg = new EmailMessage();
  msg.From = "test@email.com";
  msg.To = "support@client.com";
  msg.Subject = "SLA Alert";
  msg.Body = "Ticket #104 requires attention.";
  msg.Send();

// 2. Fluent API Way
// The code reads like a natural language sentence.
  new EmailBuilder()
    .From("test@email.com")
    .To("support@client.com")
    .WithSubject("SLA Alert")
    .WithBody("Ticket #104 requires attention.")
    .Send();
```

Listing 2.1: Comparison between Imperative and Fluent API design

By embedding the configuration logic directly within the application code, a Fluent API enables dynamic and conditional resource definitions that are difficult to achieve with static files. This promotes a more cohesive development experience; the developer defines the "Desired State" of the infrastructure within the same codebase as the application logic. This abstraction allows the developer to focus on *what* the infrastructure should provide (e.g., a persistent storage volume) rather than *how* the underlying container runtime or cloud provider provisions it.

2.3.2 Observability and OpenTelemetry

Observability is the ability to understand a system's internal state by looking at its external outputs. In complex distributed systems, observability is a more advanced

and thorough form of traditional monitoring. While monitoring usually checks predefined thresholds to determine if a system is working properly, observability tries to uncover the "unknown-unknowns". It provides the context needed to understand why a system behaves in a specific way. To achieve this broad view, an application must produce high-quality telemetry data categorized into three main areas: logs, metrics, and traces.

Logs provide the most granular level of detail, serving as timestamped records of discrete events that occur within a service. These are precious for post-mortem debugging and understanding the exact sequence of events leading to a specific failure. Metrics complement this information, which are numerical aggregations measured over time. Metrics provide a high-level overview of system health and performance trends, such as CPU utilization, memory consumption, or request throughput, allowing for the proactive detection of anomalies. However, in a microservices architecture, logs and metrics alone are often insufficient because a single user request may traverse dozens of independent services. Distributed tracing addresses this by following the path of a request through every component it touches, providing a visual map of the entire transaction. By correlating these traces across service boundaries, developers can identify hidden bottlenecks and pinpoint the exact source of latency in a complex call chain.

To collect and export this diverse data without getting locked into a single vendor, the industry has settled on OpenTelemetry (OTel) [31] as the standard. OpenTelemetry is a vendor-neutral framework that offers a consistent set of APIs, SDKs, and tools to instrument applications across various programming languages. By using the OpenTelemetry Protocol (OTLP), a centralized "Collector" can gather telemetry data and send it to different analysis and visualization platforms. This standardization enables organizations to switch between different observability tools without having to rework their entire application code. In the .NET ecosystem, this is done using the native Activity and DiagnosticSource classes, which support efficient telemetry collection that fits perfectly with the larger OpenTelemetry framework.

2.3.3 Identity and Security

In the context of distributed systems, managing identity and security is a multifaceted challenge that is essential for protecting sensitive data and ensuring that resources are accessed only by authorized entities. While traditional monolithic architectures often rely on stateful, session-based authentication (where a user's iden-

tity is stored in a centralized server-side session and tracked via a browser cookie) this approach is inherently incompatible with the requirements of microservices. In a decentralized environment, services are frequently scaled and distributed across multiple nodes, making the maintenance of a shared session state a significant bottleneck and a potential single point of failure. Consequently, modern cloud-native applications have transitioned toward federated identity and token-based authentication mechanisms to ensure scalability and robustness.

This paradigm shift necessitates a clear distinction between authentication (AuthN), which verifies the identity of a user or service, and authorization (AuthZ), which determines the specific actions an identified entity is permitted to perform. To achieve this in a distributed manner, the system must adopt protocols that support "location transparency" for security. Instead of each service maintaining its own user database, security logic is offloaded to a dedicated Identity Provider (IdP). This provider issues digitally signed artifacts, typically JSON Web Tokens (JWTs), which carry identity claims and authorization scopes. Because these tokens are self-contained and cryptographically verifiable, any microservice within the ecosystem can independently validate the caller's identity without needing to perform a synchronous call back to a central database or the IdP, thus preserving the system's performance and autonomy.

Two key technologies serve as the cornerstone of this security architecture: OpenID Connect (OIDC) [29] and OpenIddict [30]. OpenID Connect is an interoperable authentication layer built on top of the OAuth 2.0 framework. While OAuth 2.0 was originally designed for delegated authorization, OIDC extends this by introducing the ID Token, a standardized format that allows "Relying Parties" (applications) to receive verifiable assertions about an end-user. Complementing this standard is OpenIddict, a versatile, .NET-native framework designed to simplify the implementation of OIDC servers. OpenIddict provides the necessary infrastructure to manage the complex lifecycle of tokens, including issuance, validation, and revocation, while integrating seamlessly with the ASP.NET Core ecosystem and Entity Framework Core for persistence. By leveraging these technologies, the system can support advanced features such as Single Sign-On (SSO) and fine-grained access control across diverse platforms. As the following chapters will demonstrate, these components are instrumental in the development of the use-case platform, where they provide the underlying security fabric required for multi-tenant service interaction.

2.4 The Development Ecosystem

The implementation of this thesis leverages the .NET ecosystem, which has evolved from a Windows-centric framework into a high-performance, cross-platform foundation for cloud-native workloads. The selection of this stack is not merely a matter of language preference but a strategic choice to utilize a unified execution model. The following subsections examine the specific components of this stack, illustrating how they provide a consistent programming model for data persistence, security, and UI design.

2.4.1 .NET Technologies

.NET 10 [9] is the latest evolution of Microsoft's cross-platform, open-source development framework released in November 11th 2025. Its cloud-native optimizations, particularly in the realms of Native Ahead-of-Time (AOT) compilation and reduced memory footprints are critical for microservices, as they enable faster startup times and lower resource consumption in containerized environments.

Operating atop this runtime is ASP.NET Core, a high-performance framework specifically engineered for building modern, internet-connected applications such as web APIs and microservices. Its modular middleware pipeline and built-in dependency injection container provide the necessary extensibility to handle complex request-processing workflows while maintaining industry-leading throughput.

2.4.2 Kestrel

Kestrel [23] is an open-source web server for ASP.NET Core applications, designed to be lightweight, high-performance, and cross-platform. It serves as the default web server for ASP.NET Core and is optimized for handling HTTP requests in a cloud-native environment. It supports common web protocols, including HTTP/1.1, HTTP/2, HTTP/3 and WebSockets, allowing it to efficiently manage a wide range of web traffic scenarios. Kestrel's architecture is built around an asynchronous I/O model, which enables it to handle a large number of concurrent connections with minimal resource consumption. This makes it particularly well-suited for microservices, where scalability and responsiveness are paramount. Additionally, Kestrel can be used in conjunction with a reverse proxy server (such as Nginx or IIS) to provide additional features like load balancing, SSL termination, and enhanced security.

2.4.3 Blazor

Blazor [20] is a sophisticated, open-source UI framework that allows for the development of web interfaces using C# instead of JavaScript. It represents a significant shift in web development paradigms, enabling developers to leverage their existing C# skills to build interactive client-side applications. With it, it is possible to create reusable UI components using HTML, CSS and .NET code, which can be shared across different parts of the application. One of it's main characteristic is the ability to run in different environments: Blazor Server, where the UI logic executes on the server and updates are sent to the client via SignalR; and Blazor WebAssembly, where the application is compiled into WebAssembly and runs directly in the browser.

This approach allows for a unified development experience, often referred to as "full-stack C#", where data models and validation logic can be shared seamlessly between the server and the client. This unification reduces the cognitive load on developers and minimizes the discrepancies that typically arise when maintaining separate codebases for different layers of the application.

2.4.4 Entity Framework (EF)

Data persistence and management are handled through Entity Framework Core (EF Core) [21], a modern, lightweight Object-Relational Mapper (ORM). EF Core serves as an abstraction layer that bridges the conceptual gap between the object-oriented domain models in .NET and the structures of the underlying database (relational or non-relational). By allowing developers to interact with data using Language Integrated Query (LINQ) [24], the framework enhances productivity and ensures type safety during data operations. Furthermore, its provider-agnostic architecture ensures that the application remains decoupled from specific database engines, facilitating a more flexible infrastructure where the storage backend can be swapped or migrated with minimal impact on the business logic.

2.4.5 Swashbuckle

Swashbuckle [4] is a powerful library that integrates seamlessly with ASP.NET Core which automatically generates Swagger [36] (OpenAPI) documentation for the ASP.NET Core services. This integration is vital in a microservices context, as it provides a standardized, interactive interface for exploring and testing API endpoints. Beyond documentation, it enables the automated generation of client SDKs,

ensuring that service consumers are always synchronized with the latest API contracts.

2.4.6 TabBlazor and Tabler

TabBlazor [38] is a Blazor component library that implements the design system of Tabler [40]. Tabler provides a robust, responsive dashboard template built on modern CSS principles, ensuring that the platform remains accessible across a wide range of devices and screen sizes. This aesthetic foundation is implemented via TabBlazor, a component library that wraps the Tabler design language into reusable Blazor components. By utilizing this component-based architecture, the platform achieves a consistent "look and feel" across different modules while significantly accelerating the development of complex UI elements, such as data tables, charts, and interactive forms. This synergy between a standardized design system and a C#-native UI framework ensures that the platform is both visually professional and technically maintainable.

2.4.7 Polly

Polly [32] is a high-performance, transient-fault-handling library for the .NET environment and a member of the .NET Foundation. It provides a fluent, thread-safe API that allows developers to implement defensive programming patterns, ensuring applications remain resilient in the face of the inherent instabilities of distributed networks and microservices.

In a modern architecture, a failure in a single downstream component can lead to a cascading system collapse. Polly mitigates this risk by wrapping execution logic in Resilience Pipelines. These pipelines allow for the composition of multiple strategies to handle latency, temporary outages, or resource throttling.

Core Resilience Strategies:
Polly offers a variety of built-in strategies that can be used individually or in combination:

- **Retry**: Automatically re-executes an operation if it fails, which is ideal for overcoming self-correcting, temporary issues.
- **Circuit Breaker**: Temporarily halts requests to a failing service to prevent system exhaustion and allow the downstream resource time to recover.

- **Timeout**: Sets a maximum duration for an operation, ensuring that slow responses do not tie up threads and resources indefinitely.
- **Rate Limiter**: Controls the throughput of requests to stay within the limits of a resource, preventing penalties and crashes due to overloading.
- **Fallback**: Provides a "Plan B" (e.g., returning cached data or a default value) when an operation ultimately fails, maintaining a seamless user experience.
- **Hedging**: Executes multiple concurrent attempts of the same operation and adopts the result of the fastest successful response to minimize latency.

Ecosystem and Extensibility:
Beyond basic fault handling, Polly supports advanced scenarios through integrated Telemetry and Monitoring, allowing for deep analysis of pipeline performance. It is designed for seamless Dependency Injection and includes tools for Chaos Engineering, enabling developers to intentionally inject faults to verify system stability. The library is further supported by a robust community (Polly-Contrib) and is a foundational component in Microsoft's reference architectures, such as the eShop [22] microservices sample.

2.4.8 Visual Studio

Visual Studio [26] serves as the primary Integrated Development Environment (IDE) for this thesis, providing a comprehensive ecosystem designed to manage the full complexity of the software development lifecycle. Beyond simple code editing, it functions as a centralized command center that facilitates deep debugging, profiling, and testing of distributed systems. Its native integration with .NET 10 allows for seamless navigation through large-scale solutions, while advanced features like IntelliSense and AI-assisted coding through GitHub Copilot significantly reduce the cognitive load associated with managing multiple microservices.

Crucially, Visual Studio bridges the gap between local development and cloud-native production environments by offering robust support for containerization and orchestration. It allows for the simultaneous execution and debugging of multiple projects, enabling developers to observe the real-time interaction between front-end interfaces, back-end APIs, and infrastructure dependencies such as databases or message brokers. This tight integration ensures that the "Inner Loop" of development remains efficient and high-fidelity, minimizing the friction typically en-

countered when transitioning code from a local workstation to a distributed cloud environment.

2.5 Supporting Infrastructure

While the application logic is defined within the .NET ecosystem, its execution and reliability depend on a robust supporting infrastructure. This layer provides the necessary services for data persistence, inter-service communication, and environment virtualization, ensuring that the system can scale and recover from failures autonomously.

2.5.1 PostgreSQL

Data persistence is managed by PostgreSQL [33], an advanced, open-source object-relational database system. Chosen for its proven reliability and performance under heavy workloads, PostgreSQL ensures strict ACID (Atomicity, Consistency, Isolation, Durability) compliance, which is critical for maintaining the integrity of CRM business entities.

The choice of PostgreSQL is also strategic for its hybrid capabilities: its native support for JSONB allows for efficient storage of semi-structured data, providing the flexibility of a NoSQL engine while maintaining the rigors of a relational system. Furthermore, the use of GIN (Generalized Inverted Indexing) ensures that these documents remain highly queryable. In this microservices architecture, the system strictly adheres to the "Database-per-Service" pattern. This architectural constraint prevents temporal coupling and "hidden" dependencies at the data layer, ensuring that each service maintains absolute authority over its schema and can be scaled or refactored independently without impacting other domains.

2.5.2 RabbitMQ

To facilitate asynchronous communication and ensure system decoupling, the architecture utilizes RabbitMQ [34] as its primary message broker, implementing the AMQP (Advanced Message Queuing Protocol). By adopting an Event-Driven Architecture (EDA), the system moves away from brittle, synchronous chains of HTTP calls which are prone to cascading failures.

Services communicate through an exchange-queue model, allowing for a sophisticated routing of events. This approach supports the Competing Consumers pattern, enabling horizontal scalability; as the load increases, additional service

instances can be deployed to process messages from the same queue in parallel. Furthermore, RabbitMQ's implementation of Message Acknowledgments and Persistent Exchanges guarantees that business-critical events (such as transactional notifications or audit logs) are durable. This ensures "eventual consistency" across the distributed system, even in the event of temporary network partitions or service downtime.

2.5.3 Docker

Docker is the industry standard for containerization. The adoption of Docker over traditional Virtual Machines (VM) is a fundamental architectural decision based on the need for efficiency and speed. While both technologies provide isolation, they do so at different layers of the stack. A VM includes not only the application and its dependencies but also a full copy of a Guest Operating System, which runs in a hypervisor. This results in significant overhead in terms of disk space, memory usage, and boot times. In contrast, Docker uses operating-system-level virtualization sharing the host system's kernel running in isolated user spaces called containers. This lightweight approach allows for rapid startup times (often in seconds) and efficient resource utilization, as multiple containers can run on a single host without the need for separate OS instances.

Docker allows developers to package an application along with its entire runtime environment including libraries, dependencies, and configuration files into a single, immutable Image. This ensures "environmental consistency", meaning the application behaves identically in development, testing, and production stages, effectively eliminating the "it works on my machine" phenomenon.

In practice, the lifecycle of a containerized service begins with a Dockerfile, a script containing the successive instructions required to build the application image. For development environments where multiple services (such as the web API, the database, and the message broker) must run in tandem, Docker Compose is utilized. It serves as a multi-container orchestration tool that defines the entire local stack in a single YAML file, allowing the complex infrastructure to be initialized with a single command.

Containerization Example: The Dockerfile

The lifecycle of a containerized service begins with a Dockerfile. This file serves as a manifest of the environment, ensuring that the application runs identically across different stages of the delivery pipeline.

```dockerfile
# Use the official .NET 10 SDK for the build stage
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
WORKDIR /src

# Copy project files and restore dependencies
COPY ["ThesisProject.Api/ThesisProject.Api.csproj", "./"]
RUN dotnet restore

# Copy the remaining source code and build
COPY . .
RUN dotnet publish -c Release -o /app/publish /p:UseAppHost=
    false

# Use the lightweight ASP.NET runtime for the final image
FROM mcr.microsoft.com/dotnet/aspnet:10.0 AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "ThesisProject.Api.dll"]
```

Listing 2.2: Dockerfile example for an API service

Local Orchestration: Docker Compose

For development, Docker Compose is utilized to define and run multi-container applications. It acts as a local orchestrator, ensuring that dependencies (such as the database and the message broker) are available before the application starts.

```yaml
services:
  api:
    build: .
    environment:
      - ConnectionStrings__DefaultConnection=Host=db;Database=
    api-db;
    depends_on:
      db:
        condition: service_healthy
  db:
    image: postgres:16-alpine
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
```

Listing 2.3: Docker Compose example for local development

Windows Subsystem for Linux (WSL)
The Windows Subsystem for Linux (WSL) is a compatibility layer developed by Microsoft that allows users to run a GNU/Linux environment (including command-line tools, utilities, and applications) directly on Windows, unmodified, without the overhead of a traditional virtual machine or dual-boot setup. While the initial iteration (WSL 1) functioned via a translation layer that mapped Linux system calls to the Windows NT kernel, the current standard, WSL 2, utilizes a highly optimized lightweight utility VM.

2.5.4 Kubernetes (K8s)

While Docker provides the packaging format, Kubernetes (K8s) [2] provides the operational intelligence required to manage these packages at scale. It serves as a cluster-level operating system that abstracts the underlying hardware, allowing the system to treat a fleet of servers as a single computational unit.

Kubernetes operates on a declarative model using manifests, these manifests describe the "Desired State" of the system, and the Kubernetes Control Plane works continuously to maintain that state.

Kubernetes Object Example: Deployment and Service
To run an application in Kubernetes, we define a Deployment (to manage the pods and scaling) and a Service (to provide a stable network endpoint).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
spec:
  replicas: 3 # Ensures three instances are always running
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: api
        image: myregistry/api:v1.0
        resources:
          limits:
```

```yaml
20              cpu: "500m"
21              memory: "512Mi"
22 ---
23 apiVersion: v1
24 kind: Service
25 metadata:
26   name: api-service
27 spec:
28   type: LoadBalancer
29   ports:
30   - port: 80
31     targetPort: 8080
32   selector:
33     app: api
```

Listing 2.4: Kubernetes sample configuration file

This configuration ensures that if one instance of the API fails, Kubernetes will automatically reschedule a new pod to maintain the replica count of three. The Service object acts as an internal load balancer, distributing incoming traffic across the healthy pods. This level of automation is what enables the self-healing and high availability characteristics discussed in the earlier sections of this thesis.

Chapter 3

Aspire

In this chapter, we explore .NET Aspire (hereafter referred to as "Aspire" following the branding shift introduced in version 13.0), an open-source tool designed by Microsoft, under the .NET Foundation, to simplify the development, orchestration, and deployment of distributed applications.

3.1 Introduction

Initially introduced at .NET Conf 2023 as a technical preview, Aspire emerged as a response to the growing complexity of the cloud-native landscape. It officially reached general availability with the release of version 8.0 during Microsoft Build 2024, positioned as an opinionated, cloud-ready stack designed to streamline the development of distributed applications. As an open-source project under the .NET Foundation, Aspire benefits from transparent governance and a community-driven evolution that ensures it remains aligned with real-world developer needs.

The primary mission of Aspire is to bridge the "Inner Loop" gap: the friction between a developer's local environment and the complex reality of cloud-native production. While modern architectures utilize microservices, containers, and managed cloud services, running these locally often requires manual script-writing and fragile, environment-specific configurations. Aspire replaces this fragmented approach with a unified toolchain that manages orchestration, telemetry, and service discovery natively.

3.2 Evolution and Versioning

Since its inception, Aspire has undergone a rapid evolutionary cycle closely synchronized with the broader .NET ecosystem. Following the stable foundation of version 8.0, Aspire 9.0 arrived in November 2024 alongside .NET 9, introducing refined components and expanded third-party integrations. However, a significant architectural pivot occurred in November 2025; to unify its lifecycle with .NET 10, Aspire bypassed intermediate version numbers to launch Aspire 13.0.

This milestone marked Aspire's transition into a language-agnostic, "polyglot" platform. The shift expanded its utility beyond the C# ecosystem, allowing it to orchestrate services written in various languages (such as Python, Go, and Node.js) while maintaining a unified dashboard and OpenTelemetry model. As of February 11, 2026, the latest stable iteration is Aspire 13.1.1. This servicing release focuses on hardening the new Model Context Protocol (MCP) integrations and ensuring seamless compatibility with the latest .NET 10 security patches. This rapid maturation underscores Microsoft's commitment to making Aspire the de facto standard for cloud-native development, far beyond the traditional boundaries of the .NET domain.

3.3 AppHost and The Code-First Paradigm

A defining characteristic of Aspire is its departure from traditional, document-based orchestration. While established tools such as Docker Compose or Kubernetes rely on static YAML or JSON manifests, Aspire adopts a Code-First orchestration model via the AppHost project. This project serves as the central "orchestration brain", utilizing a strongly-typed, general-purpose programming language to define the system's topology.

By treating infrastructure as C# code, Aspire introduces several significant software engineering advantages. First, it provides Type Safety, ensuring that configuration errors (such as a service referencing a non-existent database) are caught at compile-time rather than during a failed deployment. Second, it leverages the full power of the IDE, offering IntelliSense for real-time documentation and code completion of infrastructure dependencies. Finally, this approach promotes Environment Parity; the same C# logic used to orchestrate a database locally is utilized by the Aspire toolchain to generate the declarative manifests required for production environments. For a developer familiar with the .NET ecosystem, the syntax mirrors the standard `Program.cs` structure, effectively removing the need to master

separate configuration languages or proprietary DSLs (domain specific languages).

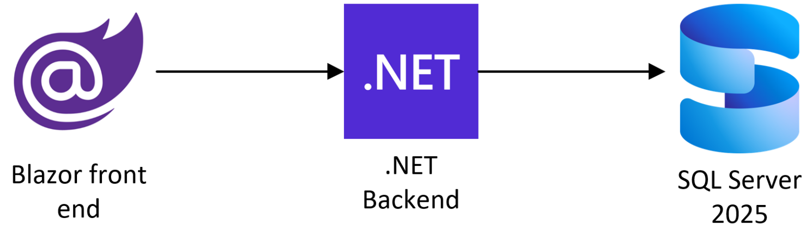


Figure 3.1: System Architecture Overview

To illustrate this paradigm, consider a standard three-tier architecture comprising a Blazor frontend, a backend Web API, and a persistent SQL Server database (as shown in Figure 3.1). The entire orchestration for this distributed system can be defined within the AppHost project as follows:

```
var builder = DistributedApplication.CreateBuilder(args);

// Resource Definition: SQL Server with persistence
var sqlserver = builder.AddSqlServer("ms-sql")
    .WithImageTag("2025-latest")
    .WithLifetime(ContainerLifetime.Persistent)
    .WithDataVolume("ms-sql-data");

// Resource Definition: API with Reference and Health Checks
var apiService = builder.AddProject<Projects.
   AspireApp1_ApiService>("apiservice")
    .WithHttpHealthCheck("/health")
    .WithReference(sqlserver)
    .WaitFor(sqlserver);

// Resource Definition: Web Frontend with External Access
builder.AddProject<Projects.AspireApp1_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithHttpHealthCheck("/health")
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Build().Run();
```

Listing 3.1: Standard Orchestration in Aspire

The first thing that a developer familiar with .NET will notice is that this structure is very similar to the typical `Program.cs` file found in .NET applications, so there is no need to learn a new configuration language or syntax.

3.3.1 Architectural Analysis of the AppHost

The structure of Listing 3.1 demonstrates the transition from imperative code to declarative orchestration. The process begins with `DistributedApplication.CreateBuilder(args)`, which initializes the orchestration engine. Unlike a standard web host, this builder does not serve HTTP requests itself; rather, it manages the lifecycle of the surrounding ecosystem.

A critical feature shown in the database definition is the use of Resource Abstractions: by calling `AddSqlServer()`, the developer does not need to manually define container images or port mappings, instead, they define a logical resource. The inclusion of `WithLifetime()` and `WithDataVolume()` addresses one of the primary "frictions" of local development: ensuring that data persists across orchestrator restarts, which is a common pain point in traditional container-based workflows.

The integration of services via the `WithReference()` method highlights Aspire's Service Discovery capabilities. When the webfrontend references the apiService, Aspire automatically injects the necessary environment variables and network coordinates. This eliminates the "Configuration Drift" mentioned in Chapter 2, as connection strings are no longer hardcoded in JSON files but are dynamically resolved at runtime. Furthermore, the `WaitFor()` method introduces a sophisticated Dependency Awareness; the orchestrator pauses the startup of the application layer until the underlying data or messaging services have passed their respective health checks. This ensures a deterministic startup sequence, preventing the "race conditions" that often cause microservices to crash during initial deployment.

3.3.2 Resource Orchestration and Lifecycle

In the Aspire ecosystem, a "resource" is defined as any discrete component that constitutes the distributed system, ranging from containerized microservices and databases to frontend interfaces. The `AppHost` acts as a declarative coordinator where developers define these resources and their interdependencies using a fluent API. At runtime, Aspire's orchestration layer manages the entire lifecycle of the system, ensuring that components are initialized, configured, and monitored in a deterministic sequence.

The orchestrator utilizes the dependency graph to launch resources in the correct order, injecting necessary metadata through a mechanism of automatic environment variable propagation. When a link is established via the `WithReference()` method, Aspire dynamically generates and provides the consuming resource with

the required connection strings and endpoint URLs. Beyond initial startup, Aspire provides continuous monitoring of resource health, enabling automated recovery actions, such as process restarts, to maintain system stability without developer intervention.

3.3.3 Service Discovery and Network Abstraction

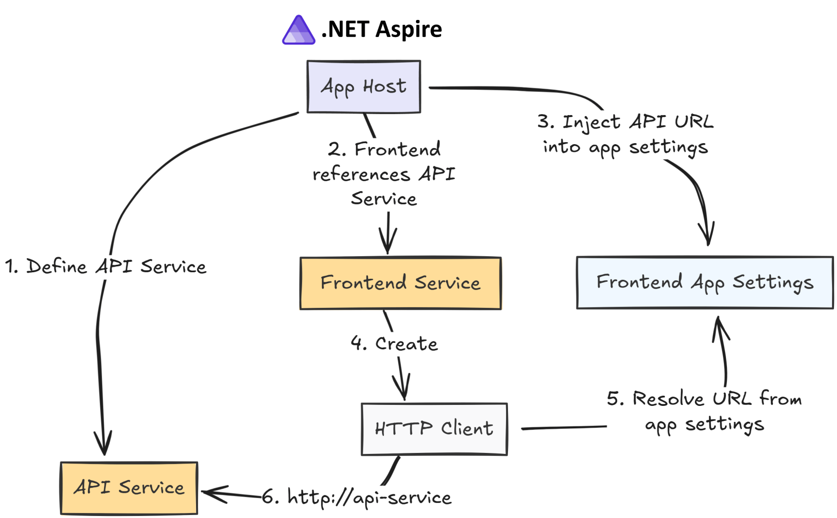


Figure 3.2: Aspire Service Discovery via Environment Variable Injection [28]

Aspire implements a sophisticated model of Implicit Service Discovery. When a reference is declared, the orchestrator bypasses the need for hardcoded IP addresses by injecting environment variables that the .NET configuration system automatically maps to logical service names (e.g., `services__apiservice__http__0`).

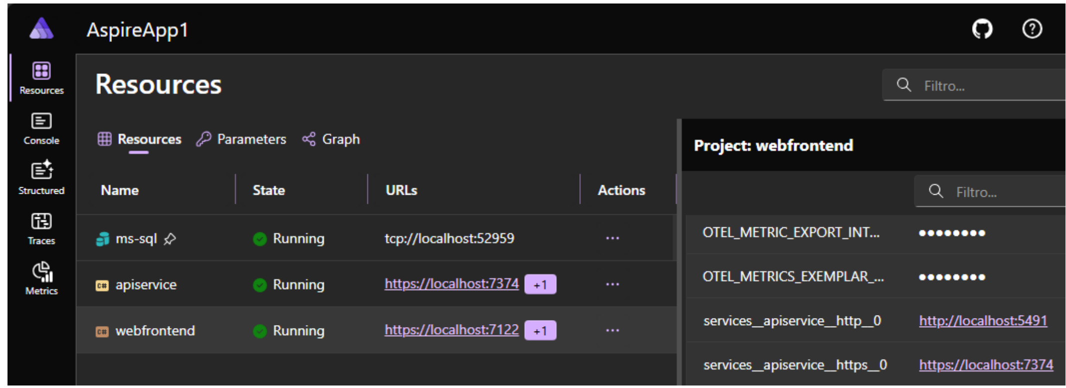


Figure 3.3: Aspire Resource Discovery Example

This abstraction ensures that the application code remains agnostic of its execution environment. As illustrated in Table 3.1, Aspire maintains functional parity

between the local development experience and production-grade orchestrators like Kubernetes.

Feature	Local Dev (Aspire)	Production (K8s)
Service Discovery	Env Variable Mapping	DNS / K8s Service
Lifecycle Management	Managed by AppHost	Managed by Kubelet
Secret Management	User Secrets	Key Vault / Secrets

Table 3.1: Comparison of Local and Production Orchestration Mechanics

3.3.4 Health Monitoring and Resource Readiness

Health checks in Aspire serve as a foundational signaling mechanism for monitoring the operational readiness of distributed components. The architecture divides this monitoring into two distinct contexts: application-level probes and orchestration-level readiness. At the application level, services expose standardized `/health` (readiness) and `/alive` (liveness) endpoints to inform load balancers of their internal state. Simultaneously, the `AppHost` utilizes these signals via the `WaitFor()` directive to manage startup sequencing. This ensures that infrastructure dependencies, such as relational databases or message brokers, are fully responsive before compute resources are initialized, effectively preventing cascading failures during the system's boot phase.

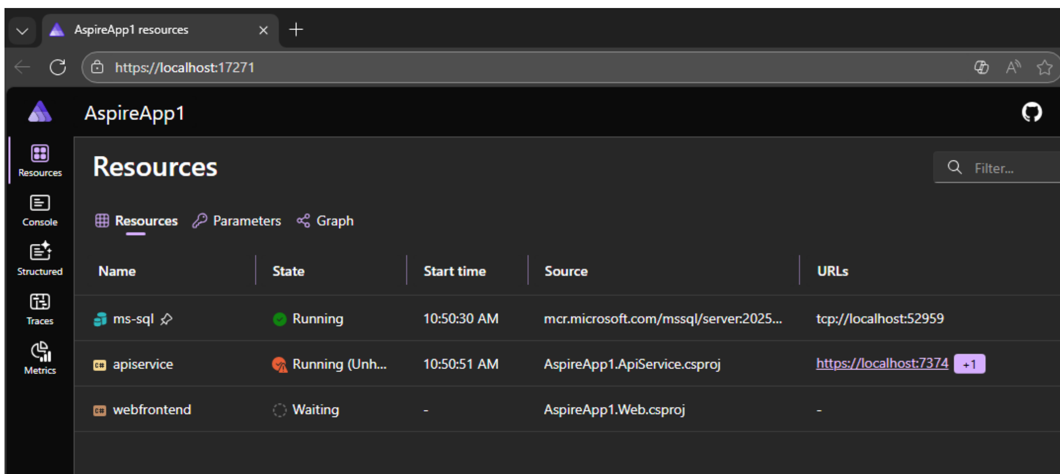


Figure 3.4: Aspire Resource Health and Readiness Overview

3.4 Service Defaults and Standardized Governance

A pivotal architectural pattern in Aspire is the delegation of cross-cutting concerns to a dedicated "Service Defaults" project. Identified with `IsAspireSharedProject` property in the project file, this library (typically named *AppName.ServiceDefaults*) serves as the centralized governance layer for the entire distributed system.

The primary entry point for this governance is the `AddServiceDefaults()` extension method. This method abstracts the inherent complexity of cloud-native infrastructure by enforcing a set of "opinionated" configurations across all participating services. By invoking this method in the `Program.cs` of each microservice, developers ensure that telemetry, health monitoring, and service discovery are implemented uniformly, preventing "configuration drift" and reducing boilerplate code.

```
public static TBuilder AddServiceDefaults<TBuilder>(this
    TBuilder builder) where TBuilder : IHostApplicationBuilder
{
    builder.ConfigureOpenTelemetry();
    builder.AddDefaultHealthChecks();
    builder.Services.AddServiceDiscovery();
    builder.Services.ConfigureHttpClientDefaults(http =>
    {
        // Turn on resilience by default
        http.AddStandardResilienceHandler();
        // Turn on service discovery by default
        http.AddServiceDiscovery();
    });
    return builder;
}
```

Listing 3.2: AddServiceDefaults Extension Method

3.4.1 Functional Components of Service Defaults

As illustrated in Listing 3.2, the service defaults project automates four critical domains of cloud-native engineering:

- **Integrated Observability:** Through `ConfigureOpenTelemetry()`, Aspire initializes logging, metrics, and distributed tracing. It automatically attaches instrumentation for ASP.NET Core, `HttpClient`, and the runtime, ensuring that every service contributes to the system's global trace map without manual intervention.

- **Operational Health:** The `AddDefaultHealthChecks()` method registers standardized liveness and readiness probes. In a development context, Aspire maps these to `/alive` and `/health` endpoints, providing the orchestrator with the signals necessary to manage service lifecycles and load balancing.
- **Resilient Networking:** A significant feature is the automatic configuration of `HttpClient` defaults. By integrating the *Standard Resilience Handler*, Aspire injects a pre-configured pipeline of retries, circuit breakers, and timeouts, aligning with the "Design for Failure" principle of distributed systems.
- **Service Discovery Integration:** By calling `AddServiceDiscovery()`, the project enables the resolution of logical service names to physical network coordinates, allowing services to communicate via abstractions rather than hardcoded URLs.

3.4.2 Extensibility and Customization

While the default template targets `net10.0` and assumes a dependency on the *Microsoft.AspNetCore.App* framework, the architecture is intentionally extensible. For non-web projects, such as background worker services or desktop integrations, developers can implement custom service defaults. This is achieved by creating a lightweight class library that mimics the `IHostApplicationBuilder` extensions, allowing even heterogeneous project types to benefit from the same telemetry and resilience policies.

Starting with Aspire 9.2 and continuing into the 13.x release cycle, the service defaults project has been further optimized to exclude health check requests from trace logs by default. This refinement ensures that high-frequency monitoring signals do not pollute the telemetry data, allowing developers to focus on meaningful application-level traces during the debugging process.

3.5 Observability and the Distributed Dashboard

A core pillar of the Aspire architecture is its native implementation of the "Three Pillars of Observability": logging, distributed tracing, and metrics. By integrating the .NET OpenTelemetry SDK, Aspire captures granular execution data that is transmitted via the OpenTelemetry Protocol (OTLP) to the integrated Aspire Dashboard.

This dashboard serves as a centralized telemetry sink, providing a real-time visualization of the distributed system. It enables the correlation of traces across process boundaries, allowing a developer to observe a single user request as it traverses multiple microservices. This high-fidelity feedback loop allows for the rapid identification of bottlenecks and failures in a way that traditional monolithic logging cannot, transforming observability from a production-only concern into a primary development-time tool.

Figure 3.5 illustrates a trace captured by the Aspire Dashboard, showcasing a request that flows through two distinct services, each contributing to the overall latency and performance profile of the operation.

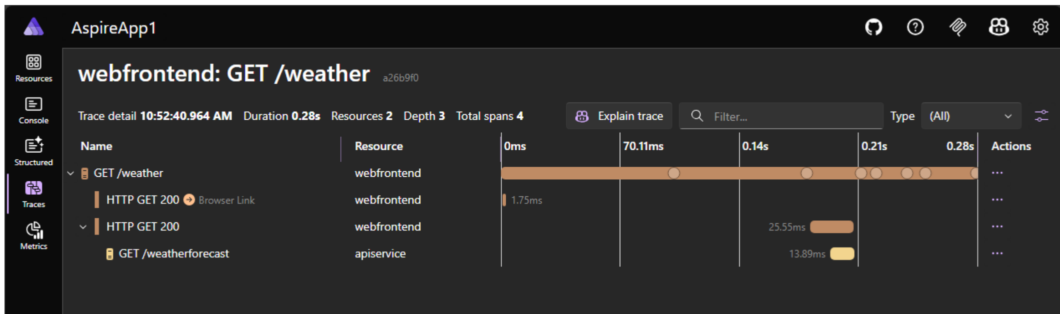


Figure 3.5: Aspire Traces Overview

Logs and metrics are similarly aggregated and visualized within the dashboard, providing a holistic view of the system's health and performance. This integrated observability model not only aids in debugging but also informs capacity planning and optimization efforts, making it an indispensable component of the Aspire ecosystem.

3.6 Networking and Service Discovery

In distributed architectures, ensuring seamless communication between services during local development is a significant challenge. Developers frequently encounter "port soup", a scenario where managing dozens of manual port assignments becomes error-prone and diverges from production networking logic. Aspire addresses these challenges by abstracting networking into a declarative model managed by the Developer Control Plane (DCP).

3.6.1 Endpoint Management and Proxies

Aspire utilizes Endpoints as the primary abstraction for service communication. These are categorized into two types:

- **Implicit Endpoints:** Automatically derived from the project's *launchSettings.json* or Kestrel configurations.
- **Explicit Endpoints:** Defined within the AppHost using the `WithEndpoint()` extension method, allowing for custom protocols and port specifications.

A critical architectural feature is the Automated Reverse Proxy. For every service binding, Aspire launches a lightweight proxy that listens on a consistent "Host Port". This proxy routes traffic to the actual service instance, which typically runs on a dynamically assigned "Random Port". This decoupling allows for seamless service replication (via `WithReplicas()`) and ensures that service-to-service communication remains stable even if underlying instances are restarted on different ports.

3.6.2 Container Networking

When orchestrating containerized resources, Aspire manages a dedicated Container Bridge Network. This virtual network provides a localized DNS server, enabling containers to resolve each other by their resource names (e.g., a web frontend connecting to `http://apiservice`).

Network Lifetimes

Aspire distinguishes between two network lifecycles:

- **Session Networks:** Temporary networks that are destroyed when the AppHost process terminates. (named `aspire-session-network-...`)
- **Persistent Networks:** Networks that remain active to support resources with persistent lifetimes, ensuring that databases or brokers remain accessible between multiple debugging sessions.

3.6.3 Configuration Sources: Launch Profiles and Kestrel

Aspire orchestrates networking by ingesting existing .NET configuration standards, ensuring that the development experience remains familiar to ASP.NET Core developers:

Launch Profiles

When `AddProject` is called, the AppHost read the *Properties/launchSettings.json* file. It selects a profile based on explicit arguments or environment variables (e.g., `DOTNET_LAUNCH_PROFILE`), using the defined `applicationUrl` to bootstrap the necessary endpoints.

Kestrel Integration

For services requiring specific server-level configurations, Aspire supports direct Kestrel endpoint definitions from *appsettings.json*. This allows developers to exclude standard launch profiles in favor of specific HTTPS or gRPC configurations, which Aspire then registers within the broader distributed application model.

3.6.4 Container Port Mapping

For third-party containers (e.g., Redis, PostgreSQL), Aspire manages the mapping between the host machine and the container's internal network. By using the `targetPort` parameter, developers can map a standard host port to the container's internal listening port:

```
builder.AddContainer("frontend", "mcr.microsoft.com/dotnet/samples")
.WithHttpEndpoint(port: 8000, targetPort: 8080);
```

This ensures that while the container internally listens on port 8080, the developer and other services can consistently access it via port 8000, maintaining a predictable interface across the ecosystem.

3.6.5 Service Discovery Mechanics

Service discovery in Aspire is logical name-based. When a resource is referenced (e.g., `WithReference(api)`), Aspire injects connection metadata into the consumer's environment via the `services__` prefix. The consumer uses the resource's name as the hostname (e.g., `http://apiservice`), which is resolved at runtime to the correct proxy URI by the .NET Service Discovery handler.

3.7 Aspire Components and Integrations

Aspire provides a dedicated ecosystem of specialized NuGet packages, known as Aspire Components, designed to bridge the gap between application logic and external infrastructure.

These components act as standardized wrappers that automate three primary engineering concerns. First, they provide Configuration Automation by binding directly to the telemetry and connection strings defined in the orchestration layer. Second, they introduce built-in Resilience Policies, such as pre-configured retry patterns and circuit breakers. Finally, they provide Native Diagnostics, as they are pre-instrumented with OpenTelemetry to ensure immediate visibility into database queries, message bus latency, and cache hits without additional manual coding.

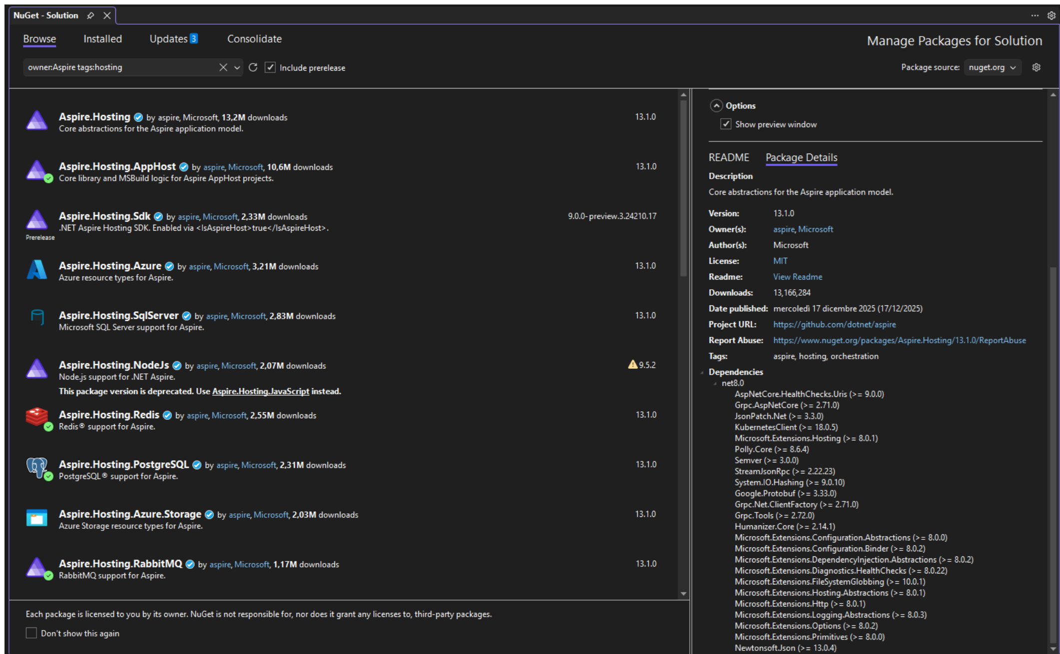


Figure 3.6: Aspire Resource Components Overview

3.7.1 Polyglot and Multi-language Support

While Aspire was initially conceptualized as a specialized orchestration stack for the .NET ecosystem, the release of version 13.0 [13] marked a significant architectural pivot toward a polyglot paradigm. This evolution transformed Aspire from a framework-specific tool into a universal cloud-native orchestrator capable of managing heterogeneous services written in a variety of programming languages, including Python, JavaScript (Node.js), and Java.

This multi-language support is achieved by decoupling the orchestration engine from the .NET runtime, allowing developers to define non-.NET resources within the same C# `AppHost`. By utilizing the `AddNodeApp()` or `AddPythonProject()` extensions, the orchestrator can manage the lifecycle, environment variables, and service discovery for these external components as first-class citizens. For a modern application architecture, this flexibility is paramount; it allows for the integration of

specialized microservices (such as a Python-based machine learning module for sentiment analysis or a React-based frontend) under a single, unified development and observability umbrella. Consequently, Aspire now serves as a holistic integration layer that harmonizes disparate technology stacks into a cohesive distributed system.

3.8 Deployment Strategies

The transition from local development to production-ready infrastructure in Aspire is defined by a rigorous separation of concerns. By decoupling the generation of infrastructure specifications from the execution of deployment logic, Aspire provides a flexible, platform-agnostic pipeline suitable for modern DevOps workflows.

3.8.1 The Specification-Execution Split

The Aspire deployment model is centered on two distinct phases, managed via the Aspire CLI:

- **Publishing (Specification Phase):** The `aspire publish` command transforms the high-level application model defined in the `AppHost` into intermediate, parameterized artifacts. These manifests (e.g., Docker Compose, Kubernetes, or Bicep) define the structural "shape" of the system while maintaining environmental neutrality.
- **Deployment (Execution Phase):** The `aspire deploy` command ingests these artifacts, resolves the associated parameters (such as secrets and connection strings), and applies the configuration to the target compute environment.

3.8.2 Hosting Integrations and Compute Environments

The system's versatility is derived from Hosting Integrations, modular NuGet packages that inject platform-specific semantics into the resource graph. This allows Aspire to support Heterogeneous Deployments, where different services within a single solution are mapped to different targets.

Through the `WithComputeEnvironment` extension, developers can explicitly disambiguate the deployment destination for specific resources. This capability is vital for hybrid architectures, such as a solution utilizing Kubernetes for backend processing while hosting the frontend on a managed container service.

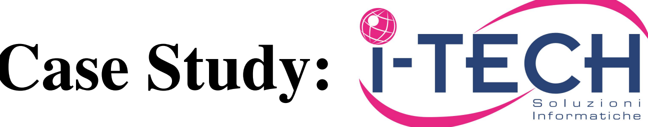
3.8.3 Parameterization and Security

A core security feature of the Aspire pipeline is the use of unresolved placeholders. Published artifacts do not contain sensitive data; instead, they utilize placeholders (e.g., `${DB_PASSWORD}`) that remain unpopulated until the execution phase. This ensures that the published manifests are immutable and safe for storage in version control or CI/CD registries, as secrets are injected only at the moment of deployment by the target environment's provider.

3.8.4 Transitioning from Legacy Manifests

With the release 9.2, Aspire has moved away from the static, JSON-based "deployment manifest" toward a dynamic API-driven model. While the legacy manifest remains available for backward compatibility and diagnostic auditing, it is no longer the primary contract. The modern approach emphasizes Extensible Annotations, such as `PublishingCallbackAnnotation`, allowing for bespoke logic to be injected into the deployment lifecycle without breaking the standard orchestration flow.

Chapter 4

Case Study: 

Now that we have explored the theoretical foundations of modern software architectures and Aspire, we turn our attention to a practical case study. This chapter details all the design and implementation choices of a distributed Customer Relationship Management (CRM) system for I-Tech S.r.l.. This system serves as a real-world testbed to evaluate how Aspire can alleviate the common frictions associated with cloud-native development, especially considering that this represents the company's first transition into distributed systems.

4.1 Introduction

I-Tech S.r.l. [7] is a small Italian software company based in Mussolente, specializing in customized Enterprise Resource Planning (ERP) solutions. With a team of ten employees, the company has established itself as a key reseller and integrator for the Mago ERP [45] ecosystem (originally developed by Microarea and now part of the Zucchetti [47] portfolio) providing high-level support and training for its customers.

Historically, I-Tech's technological stack has been fundamentally shaped by the TaskBuilder Framework [46]. TaskBuilder is a specialized development environment that evolved alongside the release of Mago.net in 2001. It leverages a combination of C++ and C# to deliver high-performance, robust ERP components. Since the modern Mago4 ERP itself is built upon this framework, TaskBuilder has deeply influenced I-Tech's choice of technologies and architectural patterns for over two decades. In the early 2000s, the .NET Framework provided a stable, industry-standard platform for the desktop and server-side applications required by the ERP's Windows-centric deployment model. During this era, I-Tech's portfolio consisted primarily of Windows Services, desktop applications, and early mobile solutions

via Windows CE, complemented by web interfaces developed in ASP.NET.

As the industry transitioned toward cross-platform capabilities and modular design, I-Tech began a systematic migration through the .NET lifecycle. This evolution saw the adoption of .NET Core and eventually the modern iterations of .NET 8 and .NET 10. During this transition, the company adapted its mobile strategy from Xamarin [27] to .NET MAUI [25] and modernized its web presence using ASP.NET Core [12].

Architecturally, however, the company continued to rely on monolithic models. This approach was historically optimal given the on-premises nature of their client base and predictable data loads. However, the increasing demand for high availability, elastic "Cloud-Native" capabilities, and external accessibility has necessitated a shift beyond these traditional boundaries. The challenge now lies in transitioning this consolidated expertise into a distributed, microservices-oriented paradigm without losing the robustness established over years of monolithic development.

4.2 State of the Art

Beyond its primary development activities, I-Tech's core business operations revolve around providing high-quality support services to its extensive customer base. To manage these mission-critical operations, the company has historically relied on a CRM system that, while functional for a smaller scale, became increasingly inadequate in the face of modern enterprise demands.

4.2.1 Architecture and Deployment

The implementation of a dedicated CRM system was not a novel concept for I-Tech S.r.l.. In the mid 2010s, as the customer base expanded, the necessity for a centralized platform to manage support tickets became a mission-critical priority. At the time, the company operated at half its current scale, and the decision to adopt a third-party solution rather than pursuing in-house development was driven by a pragmatic trade-off: prioritizing rapid time-to-market over long-term architectural alignment and technical autonomy.

The Build vs. Buy Dilemma

The decision to "buy" a third-party CRM was a strategic move driven by requirement discovery. As this was the team's first experience with a CRM, functional needs were not yet fully defined; purchasing an off-the-shelf solution allowed the

team to explore their workflows without the risk of over-engineering a custom system. Furthermore, I-Tech lacked the internal expertise to build a complex relational system from scratch, and the business case did not yet justify the high OpEx of a dedicated development team. While the team recognized that a divergent LAMP stack (Linux, Apache, MySQL, PHP) might complicate future interoperability, the immediate benefit of a ready-to-use feature set outweighed long-term integration concerns.

Stability and Customization

Over the last decade, I-Tech heavily customized this software to align with its proprietary support workflows. The most significant modification was the integration with the Mago4 ERP system to synchronize customer metadata and support history. This integration transformed the CRM into the operational backbone of the company, centralizing several critical functions:

- Ticket Management: The lifecycle tracking of customer issues from opening to resolution.
- Resource Scheduling: Centralized calendar management for technical interventions.
- Reporting: Generation of formal intervention reports in PDF format for client signatures.
- Mago4 Integration: Bidirectional data flow to ensure consistency between the CRM and ERP systems.
- Telephony Integration: Direct call initiation and tracking through the support interface (integrating an Asterisk-based PBX system).

4.2.2 Monolithic Limitations and Performance Constraints

While the legacy CRM initally provided a functional solution, its monolithic architecture introduced several critical limitations.

Limitations of the LAMP Monolith

When the CRM was initially adopted, the monolithic architecture was not perceived as a significant drawback. However, once the system became deeply embedded in daily operations, the inherent limitations of this design became increasingly apparent. The codebase was tightly coupled, with no clear separation of concerns

between different functional areas: the business logic, database queries, and HTML rendering were tightly coupled within the same PHP files. This code structure meant that a minor change in the UI could inadvertently trigger a bug in the data processing layer.

The Manifestation of Technical Risks

The core codebase was managed externally and subject to independent maintenance cycles. Because the platform lacked a modular or "plug-in" architecture, any bespoke modifications or integrations made directly to the core system were susceptible to being overwritten or causing regressions during an update.
This risk was partially mitigated for UI-level modifications, such as custom fields and forms, which were managed through the CRM's internal customization engine and persisted within the database. However, all server-side logic and core codebase extensions remained highly vulnerable. Should an upstream update modify a customized file, the local changes would be lost, necessitating a manual and labor-intensive re-implementation. This created a substantial maintenance bottleneck: every update required a comprehensive audit to identify and restore broken customizations, a process that was both time-consuming and prone to human error. While the relative infrequency of the CRM's update cycle and the team's diligent documentation helped manage this burden, the lack of a clean separation between core logic and custom extensions significantly increased the system's technical debt.

Performance Bottlenecks and Resource Contention

As the volume of support tickets and concurrent user sessions increased, the monolithic architecture exhibited significant vertical scaling limitations. System response times degraded during peak loads, primarily due to contention within the shared MySQL database. Complex join operations and high-frequency data retrieval in the ticketing module resulted in increased I/O wait times and noticeable application latency.

These issues were further compounded during synchronization cycles with the Mago4 ERP system. The synchronous nature of these integrations induced additional CPU overhead and database locking, leading to frequent request timeouts and a degraded Quality of Service (QoS) for internal agents.

Architectural Coupling and Technical Debt

The tightly coupled codebase introduced significant architectural fragility. The lack of modular boundaries meant that implementing modern requirements such as a

decoupled Customer Portal, automated SLA logic, or IMAP integration required invasive modifications to the core domain logic. This high degree of "inter-module dependency" increased the risk of regression and extended the development lifecycle (Lead Time), hindering the adoption of agile delivery practices.

Observability and Opaque Failure Modes

A critical deficiency in the legacy system was the absence of a structured observability framework. Debugging was a reactive process characterized by "black-box" execution; without distributed tracing or centralized log aggregation, root-cause analysis required manual correlation of disparate logs across system layers. This increased the Mean Time to Repair (MTTR) for production incidents.

Deployment Constraints and Scaling Inefficiency

The monolithic structure enforced a "single-unit" deployment model, where even granular security patches necessitated a full application restart. This resulted in avoidable service downtime and prolonged maintenance windows. Furthermore, the inability to scale specific high-traffic modules independently (horizontal scaling) proved inefficient for handling the "bursty" traffic patterns inherent in public-facing portals.

Technological saturation

Ultimately, the legacy CRM reached a technological saturation point. The infrastructure tax required to maintain and extend the PHP monolith began to outweigh the benefits it provided. This forced a strategic pivot: rather than continuing to patch a rigid system, I-Tech committed to a greenfield rewrite using a cloud-native, distributed architecture.

4.3 Project Goals

The decision to redesign the CRM as a distributed system was not merely a functional necessity but a strategic opportunity to modernize the company's technical stack. The primary objective was to achieve Architectural Modernization, replacing the rigid, third-party legacy monolith with a bespoke, microservices-based solution. This ensures that high-traffic components, such as the Customer Portal, can scale independently of internal administrative tools, providing a responsive experience for clients without over-provisioning resources for the entire system.

A central goal was to ensure that the move to a distributed landscape would not result in a net loss of developer productivity, but rather yield an increase in velocity through sophisticated automation. This is complemented by a commitment to Standardized Observability, moving away from the "black-box" nature of the legacy PHP system, offering real-time distributed tracing to rapidly identify bottlenecks in complex workflows.

Furthermore, the project sought to create a Unified Ecosystem to maximize code reusability. This reduces the surface area for bugs and minimizes the cognitive load on I-Tech's small development team. Ultimately, these technical goals serve the overarching business aim of deep ERP Integration, facilitating a seamless bidirectional data flow between the CRM and the Mago4 ERP to enable advanced features like real-time "Hour-Bank" tracking and automated intervention reporting.

4.4 Design Requirements

The transition to the new CRM followed the principles of Domain-Driven Design (DDD). By defining specific Bounded Contexts, the architecture ensures that each microservice manages a distinct domain of business logic, thereby achieving high fault tolerance and minimizing the "side-effect" bugs common in the legacy monolith.

4.4.1 Functional Requirements

The functional scope was divided into the preservation of core business logic and the introduction of advanced cloud-native capabilities.

Core Requirements and Legacy Preservation

The new architecture must seamlessly maintain the fundamental operations of I-Tech's support workflow. This includes the end-to-end management of support tickets, scheduling through calendar integration, and the ability to quick start a new phone call through the CRM interface. A critical component is the Reporting and PDF Generation module, which automates the creation of formal intervention reports for client signature. Furthermore, the system must handle robust user authorization and maintain deep data synchronization with the Mago4 ERP, ensuring customer data and support history are consistent across the organization.

Modernized Capabilities and New Features

To enhance operational efficiency, several new pillars were integrated into the design:

- **Self-Service Ecosystem**: A dedicated Customer Portal allows clients to track tickets and access an integrated knowledge base powered by Wiki.js [44], reducing the volume of repetitive support queries.
- **Advanced Automation**: SLA Management logic prioritizes urgent requests, while IMAP Integration ensures that customer emails are automatically converted into actionable tickets.
- **Pre-Paid Time Management**: The system introduces "Monte Ore" (Hour-Bank) Tracking, where support hours are automatically deducted from customer pre-paid hours during interventions.
- **Document Hub**: A centralized document management system handles quotations, orders, and contracts.
- **Project Management and Time Tracking**: A module to manage internal projects and track time spent on various tasks, providing insights into resource allocation and project progress.
- **Licensing and Subscription Management**: A module to collect and manage software license status and information.
- **Advanced ERP Integration**: Real-time bidirectional synchronization with Mago4 ensures that customer data, support history, and intervention reports are consistent across both systems, enabling features like automated "Hour-Bank" tracking and seamless data flow between the CRM and ERP. With the new architecture, the CRM should also be able to integrate with newer ERP systems like Zucchetti's "Mago Cloud" and "Mago Web" using the same integration layer, ensuring future-proofing and flexibility in ERP choices.

4.4.2 Non-Functional Requirements

The system's long-term success depends on several critical non-functional requirements that ensure it can evolve alongside the company's business goals. Given the diverse nature of local infrastructure, a primary objective is achieving a high degree of Environmental Adaptability. Unlike a centralized cloud service, the solution must be capable of operating within diverse and often restrictive infrastructure

environments. Therefore, the architecture must allow for the dynamic configuration of networking, external dependencies, and storage providers through a standardized interface, ensuring that the software can be deployed across different sites without requiring custom code modifications.

Observability is a non-negotiable requirement. The system must include integrated, standardized telemetry, specifically distributed tracing and structured logging. This is essential for remote troubleshooting and reducing the Mean Time to Recovery (MTTR) when incidents occur in isolated environments where the development team lacks direct infrastructure access.

Scalability and Availability are managed through a modular, decoupled design. By ensuring that core services, such as Ticketing and Notifications, are horizontally scalable, the system can adapt to the specific performance demands of different deployment scenarios. This decoupling also ensures that failures in secondary modules, such as the Reporting engine, do not lead to a total system outage, maintaining the integrity of core operations.

The requirements for Security and Maintainability are tailored for secure, local deployment. The use of Federated Identity via OpenID Connect (OIDC) allows the software to integrate seamlessly with existing local identity providers, such as Active Directory. From a maintenance perspective, the architecture minimizes "cognitive load" through a "Convention-over-Configuration" approach, which ensures that the system is easy to onboard, audit, and update, regardless of the specific local configuration.

Finally, while the current focus is on internal utility, the architecture is designed with the inherent flexibility to be transitioned into a licensable product. By prioritizing modularity and Data Sovereignty, the system is positioned as a strategic commercial asset. It ensures that sensitive data remains entirely within the client's physical and legal jurisdiction, making it a robust candidate for a wide range of external companies with strict regulatory requirements.

Chapter 5

Implementation

In this chapter, we delve into the practical development and deployment of the distributed CRM system for I-Tech S.r.l. We transition from the initial failed attempts at manual orchestration to a successful implementation leveraging Aspire and Blazor. This section details the "Inner Loop" development experience, the configuration of the AppHost, and the strategic advantages of a unified C# stack.

5.1 Technical Infrastructure and Tooling

The development environment was standardized to ensure parity across the small engineering team. Given the company's long-standing expertise in the Microsoft ecosystem, Visual Studio 2022/2026 was selected as the primary IDE, providing the necessary tooling for complex distributed debugging.

To support a containerized microservices architecture on Windows, the environment relied on Docker Desktop backed by the Windows Subsystem for Linux (WSL2). This setup provided a Linux-native kernel for running infrastructure components (such as PostgreSQL and RabbitMQ) while allowing developers to maintain a familiar Windows-based coding workflow. The integration of the .NET SDK (versions 8 and 10) served as the execution engine for the services, while the Aspire Workload was installed to manage the high-level orchestration layer.

5.2 Project Genesis and Initial Stalling

The modernization of the I-Tech CRM was initially conceived in late 2022 as a strategic transition from a legacy PHP monolith to a distributed, cloud-native architecture. The original design proposed a heterogeneous system consisting of a

React.js [8] frontend and multiple ASP.NET Core microservices, orchestrated via a collection of Docker Compose manifests. While conceptually sound, this approach quickly encountered significant operational friction that hindered development velocity.

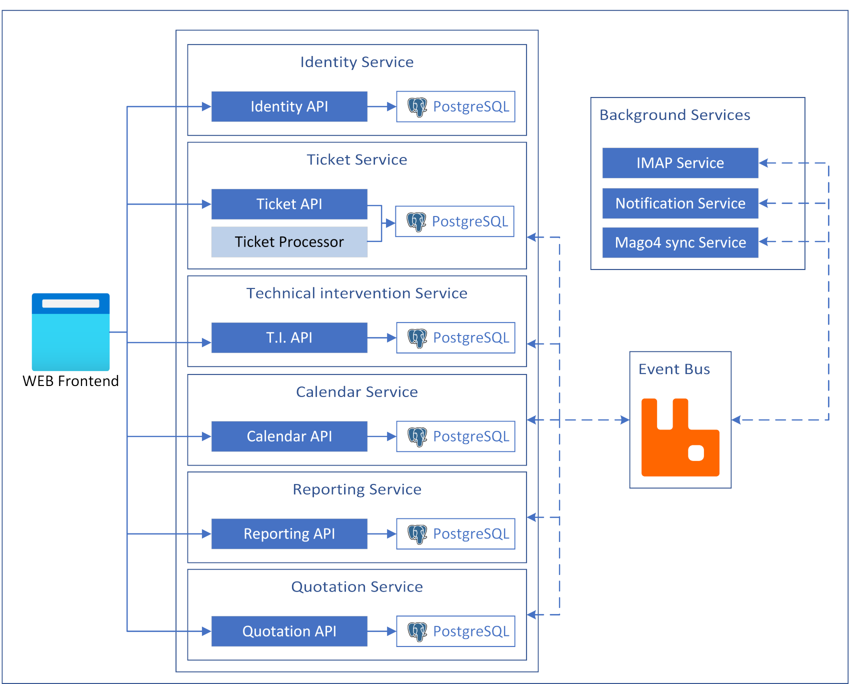


Figure 5.1: Initial architecture of the I-Tech CRM

The initial system architecture was designed with a strict separation of concerns. As illustrated in Figure 5.1, a React.js frontend communicates with a suite of ASP.NET Core microservices (Identity, Ticketing, Calendar, etc.) via RESTful APIs. To ensure modularity, each microservice was engineered to be independently deployable, maintaining its own isolated database and message broker connections. A RabbitMQ message bus was implemented to facilitate asynchronous communication, specifically supporting event-driven features such as ticket updates, automated notifications, and ERP synchronization.

However, during the transition from design to implementation, the overhead of manual orchestration and a polyglot stack introduced unsustainable complexity for a small development team. This misalignment between architectural goals and operational capacity resulted in several systemic "frictions" that inhibited progress:

- **Configuration Drift and Overhead:** Each service required manual port mapping and environment variable management. Coordinating service discovery, ensuring the frontend could reliably locate the Ticketing and Identity services across shifting container hostnames and port assignments, became a

source of constant configuration errors and "it works on my machine" inconsistencies.

- **Cognitive Load and Polyglot Boundaries:** Maintaining a decoupled React-to-ASP.NET workflow imposed a high cognitive load. The loss of cross-stack type-safety meant that tracing a single transaction across the JavaScript frontend and C# backend required navigating disparate project structures. This fragmented environment significantly throttled the "Inner Loop", the iterative cycle of coding, building, and debugging.
- **Manual Resource Provisioning:** Setting up backing services, such as PostgreSQL for persistence and RabbitMQ for message queuing, required manual Docker interventions to ensure persistent volume mounting and network bridge connectivity. These "infrastructure plumbing" tasks were performed outside of the application logic, leading to a disconnected development experience.
- **Orchestration Fragility:** Starting the local environment required a multi-step process: first, the developer had to ensure that all backing services were running (often via separate Docker Compose files), then manually start each microservice in the correct order to satisfy dependencies. This process was error-prone and time-consuming, leading to frequent "dependency hell" scenarios where one service would fail to start because another was not yet available.

These challenges culminated in a state of architectural paralysis. The operational cost of simply "starting" the local environment began to outweigh the time spent on developing core business logic. Recognizing that the team was spending more effort on infrastructure "plumbing" than on CRM features, the project was strategically suspended. The decision was made to wait for a more integrated ecosystem that could harmonize service orchestration and observability a gap that was eventually filled by the emergence of Aspire.

5.3 Aspire integration

Following the period of strategic suspension, the project was resumed in mid-2025 with the adoption of Aspire. By this phase, Aspire had matured into a robust ecosystem that addressed the specific operational frictions identified in the initial development cycle. Discovered during the research phase of this thesis, Aspire was se-

lected for its ability to abstract the complexities of container orchestration into a developer-centric, "code-first" experience.

The integration process was facilitated by native IDE support within Visual Studio 2026. By leveraging the "Add Aspire Orchestration" tooling, the development environment automatically scaffolded an AppHost project. This transformation effectively deprecated the brittle, manual Docker Compose manifests in favor of a type-safe C# environment.

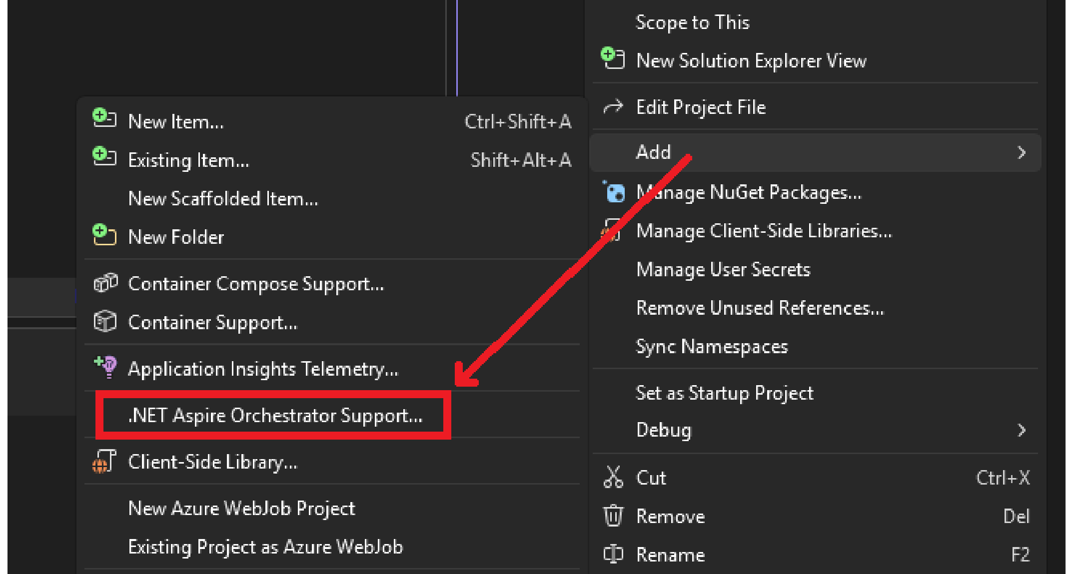


Figure 5.2: Adding Aspire Orchestration to an existing Visual Studio solution

While Visual Studio initially provisions an old stable version of the tool (such as version 9.0), the project was immediately updated to the latest release (9.x) using the Aspire CLI. This ensured access to the most recent advancements in service discovery and container management. Following the addition of necessary project references and service configurations, the AppHost assumed responsibility for the orchestration layer.

This transition represented a shift from imperative configuration (manually defining ports and networks) to declarative orchestration, where the developer defines the desired state of the system and allows Aspire to handle the underlying complexity.

5.3.1 The Unified Development Stack

During the integration of Aspire, the development team made a pivotal architectural decision. Since Aspire in versions 9.x don't yet support React.js frontends natively (the support of multiple languages was not yet implemented), the team opted for a complete shift to Blazor [20].

This tool allows developers to build interactive web UIs using C#, enabling the entire stack to be developed in a single language and it's adoption of Blazor represented a strategic alignment with the .NET ecosystem's trajectory. Choosing Blazor over React.js was not merely a reaction to early Aspire tooling limitations, but a decision to maximize architectural synergy. By utilizing a "C# Everywhere" approach, the team mitigated the risks associated with context-switching and disparate build pipelines, effectively turning a potential learning curve into a long-term productivity asset.

This unification drastically reduces the cognitive load on the development team, as they no longer need to switch between JavaScript and C# paradigms. More importantly, it enables extensive Code Sharing. Data Transfer Objects (DTOs), validation logic (using FluentValidation [6]), and business models are defined in shared class libraries, ensuring that the frontend and backend remain perfectly synchronized without the need for manual API contract updates.

Furthermore, the unified stack optimizes the debugging experience. In a typical polyglot setup, tracing a request from the UI to the database requires multiple debuggers and logs. With Aspire and Blazor, a developer can set a single breakpoint in the C# frontend and step directly into the backend API code within the same IDE session. This high-fidelity feedback loop significantly accelerated the development timeline and enabled the team to focus on business value rather than integration challenges.

5.4 Orchestration with the AppHost

The AppHost serves as the "brain" of the distributed system. Unlike traditional orchestration tools, the AppHost is a standard .NET project that utilizes a Fluent API to define service relationships. This C#-based approach eliminates the syntax errors common in YAML-based orchestration and allows for the use of standard programming constructs to manage environment-specific logic.

In the implementation for I-Tech S.r.l., the AppHost coordinates several critical components: the Blazor Web Frontend, the Identity Provider (handling OpenIddict), the Ticketing Service, and the Company Management Service. Furthermore, it manages the lifecycle of infrastructure resources, such as a PostgreSQL instance and a RabbitMQ message broker.

```
1 var builder = DistributedApplication.CreateBuilder(args);
2
3 var rabbitMq = builder.AddRabbitMQ("eventbus")
```

```
4       .WithManagementPlugin()
5       .WithLifetime(ContainerLifetime.Persistent);
6
7   var postgres = builder.AddPostgres("postgres")
8       .WithImageTag("18")
9       .WithLifetime(ContainerLifetime.Persistent);
10
11  var identityDb = postgres.AddDatabase("identitydb");
12  var ticketDb = postgres.AddDatabase("ticketdb");
13  var companyDb = postgres.AddDatabase("companydb");
14
15  // Identity Provider
16  var identityProvider = builder.AddProject<Projects.
        IT_TecnAssist_Auth>("auth-api")
17      .WithHttpHealthCheck("/health")
18      .WithReference(identityDb).WaitFor(identityDb)
19      .WithReference(rabbitMq).WaitFor(rabbitMq);
20
21  // Ticket Service
22  var ticketService = builder.AddProject<Projects.
        IT_TecnAssist_TicketService>("ticket-service")
23      .WithHttpHealthCheck("/health")
24      .WithReference(ticketDb).WaitFor(ticketDb)
25      .WithReference(identityProvider).WaitFor(identityProvider)
26      .WithReference(rabbitMq).WaitFor(rabbitMq);
27
28  // Company Service
29  var companyService = builder.AddProject<Projects.
        IT_TecnAssist_CompanyService>("company-service")
30      .WithExternalHttpEndpoints()
31      .WithHttpHealthCheck("/health")
32      .WithReference(companyDb).WaitFor(companyDb)
33      .WithReference(identityProvider).WaitFor(identityProvider)
34      .WithReference(rabbitMq).WaitFor(rabbitMq);
35
36  // Web Frontend
37  var webFrontend = builder.AddProject<Projects.IT_TecnAssist_Web
        >("webfrontend")
38      .WithExternalHttpEndpoints()
39      .WithUrls(c => c.Urls.ForEach(u => u.DisplayText = $"
        Website ({u.Endpoint?.EndpointName})"))
40      .WithHttpHealthCheck("/health")
41      .WithReference(identityProvider)
42      .WithReference(ticketService)
43      .WithReference(companyService);
```

```
44
45 // The other services that are currently missing are:
46 // - "Technical Interventions" Service
47 // - Calendar
48 // - Reporting Service
49 // - Notification Service
50 // - Imap Service
51 // - PBX Service
52 // - Documentation and FAQ Service
53 // - Quotation Service
54 // - License Service
55
56 identityProvider
57     .WithEnvironment("WEBAPP_HTTP", webFrontend.GetEndpoint("
       https"))
58     .WithEnvironment("COMPANY-API_HTTP", companyService.
       GetEndpoint("https"));
59
60 builder.Build().Run();
```

Listing 5.1: Full AppHost orchestration configuration

The primary advantage illustrated in Listing 5.1 is the high level of abstraction over operational complexity. Even for a developer with limited experience in container orchestration, the intent of the code is immediately clear. The AppHost abstracts away the complexities of Docker commands, network configurations, and service discovery. Instead, it allows developers to focus on the logical relationships between services and resources.

For instance, the configuration of the *Ticket Service* (lines 22-26) demonstrates how dependencies are modeled. Through the `WithReference` method, Aspire automatically manages Service Discovery by injecting the necessary connection strings and endpoints into the service's environment variables at runtime. This mechanism is driven by an underlying injection logic that standardizes how services communicate. As seen in the implementation of `WithReference`, Aspire dynamically encodes the `connectionName` and maps it to a structured environment variable, typically following the `ConnectionStrings__` prefix or a specific `ConnectionProperties` flag. The system ensures that metadata (such as hostnames, ports, and credentials) is consistently "splatted" into the destination container's environment.

This removes the need for hardcoded configuration files that vary between development and production environments. Then, another critical architectural feature of this configuration is the `WaitFor` mechanism. In traditional Docker Compose se-

tups, services often crash upon startup because their database dependencies are not yet "ready" to accept connections, despite the container being "running". Aspire addresses this startup race condition by orchestrating the sequence: it monitors the health of the PostgreSQL container and only initializes the dependent API services once the database is fully ready.

Additionally, the use of `WithLifetime(ContainerLifetime.Persistent)`, as we can see in line 9 of Listing 5.1, optimizes the Inner Loop of development. By allowing infrastructure containers to persist across application restarts, developers avoid the time-consuming process of re-provisioning databases or re-configuring RabbitMQ exchanges during every debug session, maintaining a stable state for iterative testing.

5.5 Infrastructure and Service Binding

A primary challenge in distributed systems is the reliable provisioning and discovery of backing services. In the legacy environment, PostgreSQL and RabbitMQ required manual installation and configuration on the host machine or were managed via static Docker Compose files with hardcoded credentials. Under the Aspire model, these external dependencies are promoted to first-class Resources within the orchestration layer.

The architecture utilizes the Service Binding pattern, where the *AppHost* manages the lifecycle, connection strings, and credentials for infrastructure such as PostgreSQL databases and Redis caches. By moving away from hardcoded connection strings in *appsettings.json*, microservices receive these sensitive details dynamically via environment variables injected at runtime. This "late-binding" approach ensures high portability; the same application logic that connects to a local ephemeral Docker container during development can seamlessly transition to an Azure Database for PostgreSQL in production without code modifications.

As demonstrated in Listing 5.1, each microservice declares its dependencies using the `WithReference` method. This does more than establish a network link; it creates a logical contract between the service and the resource. For example, the Ticket Service references both its dedicated PostgreSQL database and the Identity Provider, ensuring that the orchestrator wires the security and data layers correctly before the service begins execution.

```
1 // ---------------------------------------------------------
2 // AppHost.cs: Declarative Orchestration
3 // ---------------------------------------------------------
```

```
4  var ticketService = builder.AddProject<Projects.
   IT_TecnAssist_TicketService>("ticket-service")
5      .WithHttpHealthCheck("/health")
6      .WithReference(ticketDb).WaitFor(ticketDb)
7      .WithReference(identityProvider).WaitFor(identityProvider)
8      .WithReference(rabbitMq).WaitFor(rabbitMq);
9
10 // ---------------------------------------------------------
11 // Ticket Service Program.cs: Consuming the Bound Resource
12 // ---------------------------------------------------------
13
14 var builder = WebApplication.CreateBuilder(args);
15 builder.AddServiceDefaults();
16 builder.AddDefaultAuthentication(AuthConstants.TicketScope);
17 builder.Services.AddDbContext<TicketDbContext>(options =>
18 {
19     options.UseNpgsql(builder.Configuration.GetConnectionString
   ("ticketdb"));
20 });
21 // Omitted code for brevity
22 var app = builder.Build();
23 // Omitted code for brevity
24 app.Run();
```

Listing 5.2: Binding Ticket Service to PostgreSQL in AppHost

In Listing 5.2, the Ticket Service retrieves its PostgreSQL connection string via a logical name (`"ticketdb"`). This decouples the service from physical infrastructure details like IP addresses or specific ports, which are managed by the Aspire proxy.

Furthermore, the implementation utilizes the `AddServiceDefaults()` and `AddDefaultAuthentication()` extension methods. The former, provided by the Aspire template, automatically configures OpenTelemetry for logging and metrics, alongside Polly for resilience. The latter is a custom-defined abstraction used across the I-Tech S.r.l. ecosystem to ensure consistent JWT Bearer authentication. This encapsulates the complex OIDC configuration, allowing the development team to enforce security standards across all microservices through a single line of code, significantly reducing the surface area for configuration errors.

5.6 Security Implementation and Homogeneity

Security represents a critical pillar of the CRM architecture, especially given the sensitive nature of customer ticketing and ERP-integrated data. To address this, the system utilizes OpenIddict to implement a centralized Identity Provider (IdP) based on the OpenID Connect (OIDC) and OAuth 2.0 standards.

The integration within the Aspire AppHost proved particularly beneficial for managing the inherent complexity of the Authorization Code Flow. The AppHost acts as the "Source of Truth", dynamically injecting the *Authority* URL into every dependent microservice. This ensures that each service knows exactly where to validate incoming JSON Web Tokens (JWT) without requiring manual configuration of discovery endpoints.

To satisfy the specific multi-tenancy requirements of I-Tech S.r.l., a custom "Contextual Identity" interruption was injected into the OIDC challenge. When a user authenticates, the flow is interrupted to allow for a "SelectCompany" step. This ensures that the final Bearer token is not only bound to a user's identity but also scoped to a specific company context. Consequently, every subsequent API request is inherently tenant-aware, preventing cross-tenant data leakage a fundamental requirement for the CRM's reliability.

This technical setup directly enables Security Homogeneity across the ecosystem. By abstracting the OIDC logic into a custom `AddDefaultAuthentication()` extension method, the team ensured that every microservice, whether the Ticketing Service or the Company Service, validates tokens using identical security protocols. This reduces the likelihood of security loopholes caused by misconfiguration and ensures that as the system scales, the security posture remains consistent and easily maintainable.

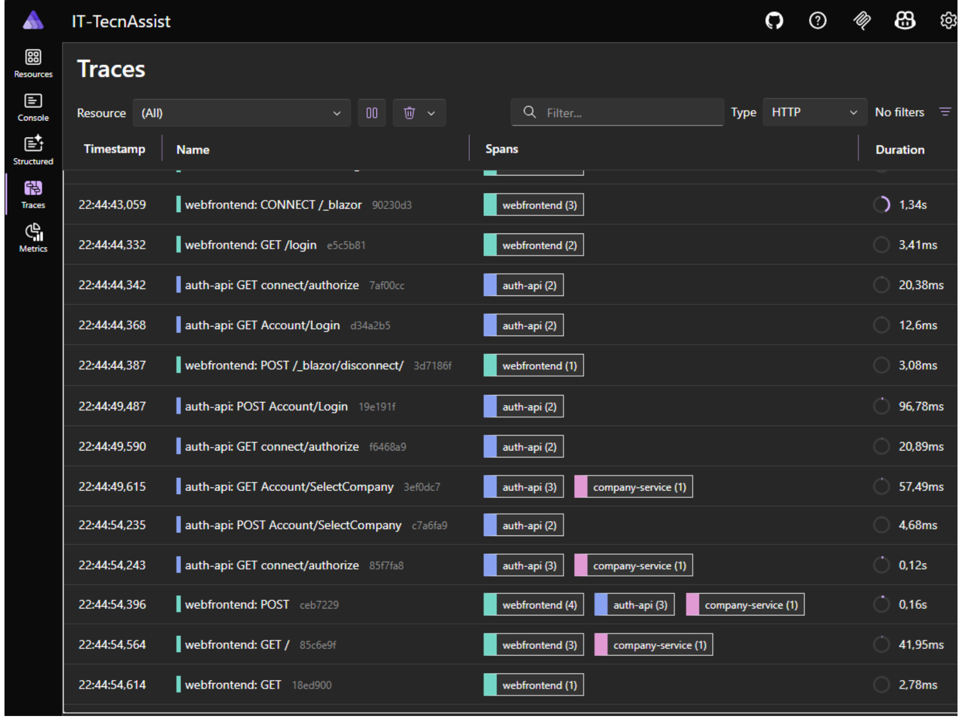


Figure 5.3: Aspire Dashboard: Distributed trace showing a successful OIDC handshake and authorized request flow across services.

As shown in the distributed traces on the Aspire Dashboard, the transparency of the authentication flow becomes immediately apparent. Developers can observe the redirect to the Identity Provider, the token exchange, and the subsequent authorized requests as they traverse the system, transforming what is usually a "black box" security process into a verifiable and debuggable sequence.

5.7 Deployment Pipeline: From AppHost to Cloud

A fundamental challenge in microservices is the "Configuration Gap": the discrepancy between local orchestration and production-grade deployment manifests. Aspire mitigates this by utilizing the `AppHost` as a Single Source of Truth. The same C# logic used to orchestrate services locally is leveraged by the Aspire CLI to generate parameterized artifacts for various hosting environments.

5.7.1 Implementation Strategy: On-Premise Docker Testing

During the development of the I-Tech CRM, the primary objective was to validate UI responsiveness and cross-service functionality. Given that the system was in its integration phase, a Docker Compose-based deployment target was selected. Us-

ing the command `aspire publish`, the application model is transformed into a standard YAML specification.

This approach allowed I-Tech to host a staging environment on an on-premise Docker server. This provided a cost-effective, high-fidelity testing ground that mirrored the local "Inner Loop" while allowing stakeholders to interact with the system remotely.

5.7.2 The Path to Production: Azure Kubernetes Service (AKS)

While the current implementation utilizes Docker, the transition to a cloud-native environment is architecturally trivial. The Aspire deployment pipeline is built on a Specification-Execution Split, which allows the same `AppHost` definition to be retargeted toward Azure Kubernetes Service (AKS) or Azure Container Apps without modifying the underlying source code.

- **Artifact Generation**: By utilizing the `Aspire.Hosting.Kubernetes` integration, the CLI generates Kubernetes manifests including *Deployments*, *Services*, and *Ingress* rules.
- **Parameterization**: Aspire utilizes a "Placeholder Principle", where sensitive data (e.g., database credentials) are generated as unresolved variables in the manifest, to be injected at runtime by Azure Key Vault or Kubernetes Secrets.
- **Environmental Parity**: This workflow ensures that service discovery logic and resource bindings are identical across all tiers, eliminating the risk of configuration-induced regressions.

5.7.3 CI/CD Integration and Extensibility

The Aspire CLI is designed to be integrated into automated pipelines (e.g., GitLab CI/CD). This allows for a GitOps workflow where every commit to the main branch triggers the `publish` command, creating an immutable snapshot of the infrastructure. For I-Tech S.r.l., this represents a significant increase in operational maturity, as the infrastructure is now versioned, audited, and deployed alongside the application code.

5.8 Summary of the Implementation

The successful implementation of the I-Tech CRM marks a transition from "Architectural Paralysis" to a state of high developer velocity. By replacing manual

Docker orchestration with the Aspire AppHost and unifying the stack with Blazor, the project eliminated the "infrastructure tax" that previously stalled development. The result is a system that is not only secure and highly observable but also maintainable by a small engineering team. This practical success serves as the foundation for the evaluation in the following chapter, where we quantify these qualitative improvements.

Chapter 6

Evaluation and Discussion

This chapter provides a critical evaluation of the impact of Aspire on the development lifecycle, system maintainability, and the overall Developer Experience (DX). By referencing specific scenarios encountered during the modernization of the I-Tech CRM, I will illustrate how Aspire's features transitioned from theoretical abstractions into tangible architectural benefits.

6.1 Observability in action: From Black Box to Transparency

The transition from a monolithic architecture to a distributed system often exacerbates the difficulty of monitoring system health. As previously discussed, the legacy PHP monolith represented a "black box" environment where debugging was a reactive and labor-intensive process. Resolving a single failed request required manual SSH access to virtual machines to locate disparate Apache or application-level logs. Without a unified trace identifier, correlating these entries was time-consuming and often inconclusive due to a lack of contextual metadata.

The "Zero-Configuration" Paradigm Shift
With the adoption of Aspire, the debugging workflow underwent a paradigm shift. During the integration of the Ticketing Service, the team encountered a 500 Internal Server Error in the Blazor UI. In the legacy environment, this would have resulted in high Mean Time to Identify (MTTI). However, using the Aspire Dashboard, the team inspected a Distributed Trace in real-time, providing a granular visualization of the request lifecycle. This transparency pinpointed the exact failure and the problem was fixed in seconds.

In a standard microservices implementation, achieving this level of insight requires significant Manual Instrumentation, including:

- Manual integration of OpenTelemetry SDKs into every discrete service.
- Deployment and maintenance of external telemetry collectors and visualizers (e.g., Jaeger for tracing or Prometheus for metrics).
- Configuration of complex networking to ensure telemetry data is exported correctly across service boundaries.

By utilizing Aspire's standardized OpenTelemetry Protocol (OTLP) exporters, these features were available "out-of-the-box". This allowed the I-Tech development team to move from a reactive debugging posture to a proactive monitoring posture, where system bottlenecks and failures are visible as they occur. Furthermore, the ability to correlate logs, metrics, and traces within a single interface significantly reduced the cognitive load on developers. Instead of juggling multiple tools, the team relied on a unified dashboard that provided both high-level overviews and low-level diagnostic details.

Technical Discussion: Abstraction vs. Mastery
While this automation acts as a powerful "Force Multiplier", it introduces a potential risk of "Observability Blindness". The abstraction of telemetry into a "dashboard-first" paradigm may lead to an over-reliance on specific tooling, potentially obscuring the underlying mechanics of telemetry collection. This highlights a critical architectural trade-off: while high-level abstractions accelerate the "Inner Loop" development, they do not replace the need for a fundamental understanding of the OpenTelemetry ecosystem.

Crucially, however, Aspire's reliance on the OTLP standard mitigates the risk of vendor lock-in. Because the telemetry generated during development is identical to that required by production-grade backends (such as Azure Monitor or Grafana), the insights gained locally translate directly into operational intelligence. This ensures that while the team benefits from a simplified interface, the system remains compliant with industry-standard observability patterns, fulfilling I-Tech's requirement for a robust, cloud-native architecture.

6.2 Maintainability and System Evolution

The long-term viability of a software system is dictated by its ability to evolve without incurring prohibitive technical debt.

In the legacy PHP monolith, I-Tech faced significant Architectural Rigidity; because the infrastructure was implicitly tied to the environment (a specific Linux VM with pre-configured Apache modules), any change to the system's topology required manual, error-prone updates.

The same was true for the initial React-to-ASP.NET attempt, where the lack of a unified stack led to "Integration Drift": the frontend and backend were developed in isolation, resulting in mismatched API contracts and duplicated domain models. This not only increased the maintenance burden but also introduced a high risk of runtime errors due to contract violations.

Aspire shifts this paradigm toward Architectural Flexibility by centralizing the system's definition.

The AppHost as Living Documentation

A recurring challenge for small teams is the onboarding of new developers and the maintenance of environment setup guides. In traditional distributed systems, this often results in extensive "README" files that quickly become obsolete. Within the Aspire-based CRM, the `AppHost` project acts as the "Living Documentation" of the entire ecosystem.

A new developer joining the team does not need to navigate an extensive infrastructure manual; the system's topology (including service dependencies, database bindings, and message broker configurations) is explicitly defined in the `AppHost`. This reduces the Cognitive Load required to understand the system, as the code itself serves as the "Source of Truth" for the architecture.

Streamlined Extensibility

The agility provided by Aspire is best illustrated by the process of extending the CRM with new business logic, such as the *Technical Interventions Service*. In a standard microservices setup, adding a service involves manually creating Dockerfiles, updating Docker Compose manifests, managing network aliases, and synchronizing environment variables across multiple files. With Aspire, this process is condensed into three high-level steps:

1. **Project Creation:** Initializing the .NET project within the existing solution.
2. **Orchestration Registration:** Adding one declarative line to the `AppHost`: `builder.AddProject<Projects.InterventionsService>("name")`.
3. **Automated Provisioning:** Aspire automatically manages the underlying networking, containerization, and injection of necessary environment variables

(such as connection strings) during development.

This level of automation ensures that the I-Tech team can focus on implementing business-critical features rather than managing the "plumbing" of a distributed environment.

Decoupling Infrastructure from Business Logic

Finally, Aspire facilitates a clean separation between Infrastructure Concerns and Domain Logic. Because services do not hardcode their connection details, but instead receive them via Aspire's resource injection, infrastructure can be swapped or upgraded with minimal impact. For instance, upgrading the PostgreSQL version or migrating from a local Redis container to a managed cloud cache involves a single modification in the `AppHost`.

This decoupling prevents Configuration Drift a common issue in the legacy system where different environments (development vs. production) had subtle, undocumented differences. By centralizing these definitions, I-Tech ensures that the CRM remains modular and capable of evolving alongside future technological shifts.

The Abstraction Trade-off: Visibility vs. Velocity

While Aspire significantly reduces the operational burden, it introduces a layer of abstraction that may obscure low-level infrastructure details. By shifting to a "dashboard-first" and declarative model, developers are effectively distanced from the generated Kubernetes manifests or the underlying container orchestration logic.

This "black box" nature of the abstraction presents a potential risk during Root Cause Analysis (RCA) in complex production environments. If an infrastructure failure occurs that falls outside the automated recovery logic, a lack of direct visibility into the underlying configuration could hinder troubleshooting efforts. Consequently, while Aspire acts as a powerful catalyst for developer velocity and system maintainability, it is imperative for the engineering team to retain a foundational understanding of container orchestration. Maintaining this balance ensures that the team can leverage high-level productivity tools without losing the technical depth required to manage the system as it evolves into higher levels of complexity.

6.3 Developer Experience (DX)

The transition from a traditional microservices setup to an Aspire-based architecture significantly influenced the day-to-day workflow of the I-Tech development team. The ability to orchestrate the entire ecosystem with a single command and debug

across service boundaries within a unified IDE session transformed the local development experience. The "Inner Loop" velocity (the time spent between code change and verifying that change in a running system) was drammatically increased.

This improvement in DX is more than a matter of convenience; it directly correlates with increased developer productivity and the maintenance of a state where technical friction does not interrupt the solving of business problems. Having experienced the lifecycle firsthand, the team categorized the DX improvements into some key areas:

- **Unified Debugging:** In the legacy React-to-ASP.NET attempt, debugging a cross-service request was a fragmented process. Developers had to manage two distinct environments and rely heavily on manual logging to trace request flows. Aspire's integration with the Visual Studio debugger allows for seamless cross-process stepping. When a developer triggers a breakpoint in the Blazor frontend, they can step directly into the Ticketing Service backend as if they were debugging a monolith. This "monolithic debugging experience" for a distributed system effectively collapses the complexity of the architecture during the development phase.

- **Reduction of Cognitive Load:** A primary source of developer frustration in microservices is the management of connection strings and service endpoints. Traditionally, this is handled via string-based environment variables in YAML files, which are prone to typos and lack compile-time validation. Aspire replaces this with Type-Safe Resource References: by using C# to define dependencies, the developer benefits from IntelliSense and compiler checks. This reduces the "Mental Mapping" required to understand how services connect, allowing the developer to focus on the logic of the service rather than the "plumbing" of the network.

- **Immediate Feedback:** The Aspire Dashboard acts as a real-time feedback mechanism. In previous iterations, verifying that a message reached RabbitMQ or that a database record was updated required switching to external management tools (e.g., RabbitMQ Management UI or pgAdmin). By centralizing logs, traces, and resource states into a single dashboard, Aspire provides immediate visual confirmation of system behavior. This immediate feedback loop is critical for a "small team" environment, where a single developer often manages multiple service boundaries simultaneously.

- **Learning Curve and Knowledge Acquisition:** The Cognitive Load of mi-

croservices adoption was substantially reduced by Aspire's "Code-First" approach. Traditionally, mastering microservices orchestration requires learning a polyglot set of tools and configurations (YAML, Docker CLI, etc). With this approach, developers can leverage their existing C# skills to define the entire system significantly lowering the barrier to entry for developers transitioning from monolithic backgrounds. Furthermore, the availability of high-quality templates (like the official *eShop* sample [22]) and documentation [17] further facilitated the learning process.

- **Inner Loop Productivity and Environmental Parity:** The complex startup of a distributed system is transformed into a Single-Click Experience. By managing the dependency graph and the orchestration of services automatically, Aspire eliminates the "Startup Latency" allowing developers to see the impact of their code changes in seconds rather than minutes.

6.4 Resilience Testing and Fault Tolerance

A core advantage of the Aspire-based architecture is the ease with which Failure Scenarios can be simulated and mitigated. By utilizing the `ServiceDefaults` project, which integrates the Polly [32] library by default, the CRM system exhibits robust fault tolerance.

During evaluation, "service down" scenarios were tested by manually stopping the *Ticketing Service* while the *Blazor UI* remained active. Because of the pre-configured exponential backoff and retry policies, the UI was able to handle the transient failure gracefully. Instead of encountering a terminal "500 Internal Server Error", the system provided a responsive state, allowing the frontend to retry the connection until the service was restored. This validates the system's ability to maintain a high level of availability even during partial outages: a stark contrast to the legacy monolith, where a single component failure often resulted in total system unavailability.

6.5 Consistency and Standardization: The Force Multiplier

In traditional microservice architectures, as the number of services grows, so does the risk of Configuration Drift, where services utilize inconsistent logging levels, retry patterns, or security configurations.

6.5.1 Standardized Cross-Cutting Concerns

The use of the `AddServiceDefaults()` extension method served as the primary mechanism for architectural enforcement. By centralizing the configuration of OpenTelemetry, Health Checks, and Resilience policies, the development team ensured that every service in the CRM ecosystem adhered to the same operational standards.

For a small team of ten, this standardization is transformative. It allows a developer to move from the "Ticketing" service to the "Identity" service and find a perfectly familiar environment. The "plumbing" of the system was abstracted into a single shared project, ensuring that high-level concerns like Circuit Breaking and Distributed Tracing were inherent characteristics of every service.

6.5.2 Security Homogeneity and Contract Safety

Implementing the `AddDefaultAuthentication()` custom extension method represents a significant advancement in system maintainability. Security configurations, particularly those involving OpenID Connect (OIDC) and OAuth2, are notoriously complex and error-prone. By abstracting the OpenIddict setup into a reusable, centralized method, I-Tech S.r.l. achieved Security Homogeneity. This ensures that every service in the ecosystem adheres to the same security headers, token validation logic, and authorization policies, drastically reducing the attack surface caused by accidental misconfiguration in individual services.

Furthermore, the shift to a unified .NET stack with Blazor has eliminated the "Contract Drift" that plagued previous polyglot attempts. Through the use of Shared Class Libraries, the system enforces Contract Safety across the entire network topology.

Any modification to a Domain Model, a Data Transfer Object (DTO), or a FluentValidation rule is immediately propagated to both the service provider and the consumer at compile-time. This "Single Source of Truth" ensures that the frontend and backend are always in synchronization, effectively eliminating the class of integration bugs that frequently occur when maintaining a separate React-based frontend and ASP.NET backend.

6.6 Cost-Benefit Analysis: The "Force Multiplier" for I-Tech S.r.l.

As with any architectural decision, the adoption of microservices comes with inherent trade-offs: the benefits of modularity, scalability, and resilience must be weighed against the increased complexity and operational overhead. For a small software house like I-Tech, the "Infrastructure Tax" of microservices can be prohibitive, consuming valuable developer time that could otherwise be spent on core business logic.

If we compare the initial development effort required to set up a microservices architecture using traditional tools (Docker Compose, manual OpenTelemetry integration, etc.) versus the Aspire paradigm, we can identify several key differences in terms of time investment and operational overhead.

- **Initial Setup Time:** Setting up a microservices architecture with Docker Compose typically involves writing multiple YAML files, configuring network aliases, and manually integrating observability tools. This process can take several days to weeks for a team unfamiliar with these technologies. In contrast, Aspire's "Code-First" approach allows for a functional microservices environment to be established within hours, as the tool handles the orchestration and configuration automatically.
- **Operational Overhead:** Maintaining a traditional microservices setup requires ongoing management of container orchestration, telemetry pipelines, and service discovery mechanisms. This can consume significant developer time and lead to "context switching" between development and operations tasks. Aspire reduces this overhead by centralizing these concerns into a single project, allowing developers to focus on business logic rather than infrastructure management.
- **Long-Term Maintenance:** The risk of Configuration Drift and integration issues increases as the number of services grows in a traditional setup. Aspire's standardized approach mitigates this risk by enforcing consistent configurations across all services through shared projects and extension methods.
- **Low level Control vs. High-level Abstraction:** While Aspire provides a powerful abstraction layer that accelerates development, it may limit low-level control over the underlying infrastructure. In scenarios where fine-tuned

performance optimizations or custom orchestration logic are required, developers may need to "escape" the abstraction and manage these concerns manually. This trade-off between ease of use and granular control is a critical consideration for teams with specific performance requirements or complex deployment scenarios.

Aspire acts as a "Force Multiplier" by significantly reducing this tax, democratizing access to distributed systems for smaller teams that lack a dedicated DevOps department and allows I-Tech to manage a sophisticated microservices architecture with the operational simplicity of a monolith. The reduction in manual orchestration and the automation of observability layers mean that the team can dedicate nearly 100% of their effort to core business logic.

6.7 Strategic Advantages of the Open-Source Ecosystem

A critical factor in the selection of Aspire for I-Tech S.r.l. was its open-source nature. Unlike the legacy CRM Aspire is developed transparently within the .NET Foundation. This transition offers several strategic advantages for a small-scale software house.

Community-Driven Evolution and Reliability
The Aspire ecosystem is supported by an active, global community of developers. This collaborative environment ensures that this tool undergoes continuous refinement based on diverse, real-world feedback. For I-Tech, this means that security patches, performance optimizations, and new resource integrations (such as support for emerging cloud providers or database engines) are delivered at a pace that exceeds proprietary alternatives. By adopting a community-backed instrument, the team leverages the collective intelligence of the industry, effectively augmenting their internal research and development capabilities.

Mitigation of Vendor Lock-in
The open-source model grants I-Tech the autonomy to inspect, modify, and extend the tool to meet bespoke business requirements. In the event that a specific infrastructure need arises which is not covered by the standard libraries, the team can contribute to the codebase or fork the relevant components to implement custom logic. This transparency eliminates the risk of Vendor Lock-in, ensuring that the

CRM's core orchestration layer remains under the firm's control, rather than being subject to the arbitrary licensing changes or deprecation schedules of an external provider.

Knowledge Accessibility

Finally, the open nature of the project facilitates easier troubleshooting and knowledge acquisition. Access to Aspire's source code on GitHub serves as the ultimate documentation, allowing developers to understand the "under-the-hood" mechanics of service discovery and resource provisioning. This transparency aligns with the goal of reducing the system's overall technical debt. Other than that, the availability of community forums, discord channels, and extensive documentation and examples further accelerates the learning curve for developers new to microservices, ensuring that the team can quickly adapt to the new paradigm without being hindered by a lack of internal expertise.

6.8 Discussion of Initial Questions

To conclude this chapter, the initial questions posed in the introduction are revisited in light of the practical implementation and evaluation of Aspire within the I-Tech CRM project. Each response is synthesized from the empirical observations gathered during the modernization process.

Q1: Impact of Orchestration on Operational Complexity and Setup Time

The adoption of Aspire resulted in a significant reduction in operational complexity by replacing imperative infrastructure management with declarative orchestration. In traditional microservices setups, adding a new service requires the manual synchronization of Dockerfiles, Docker Compose manifests, network aliases, and environment-specific configuration files.

In the I-Tech project, this multi-step process was condensed into a single "Code-First" registration within the `AppHost`. By automating service discovery and the injection of connection strings (via the `WithReference` logic), Aspire eliminated the manual "plumbing" that historically leads to configuration drift. This shift allowed the team to move from an infrastructure-centric workflow to a logic-centric one, effectively accelerating the "Inner Loop" of development and ensuring that the system's topology remains compile-time validated.

Q2: The "Dashboard-First" Observability Paradigm

The integrated observability features of Aspire transformed the debugging process from a reactive to a proactive approach. The ability to visualize distributed traces and metrics in real-time through the Aspire Dashboard provided immediate insights into system behavior, allowing developers to identify and resolve issues much more quickly than in the legacy "Black Box" environment.

This paradigm significantly reduced the Mean Time to Identify (MTTI) for failures; developers could correlate logs and traces within a single interface without the cognitive overhead of switching between disparate logging tools or manual SSH sessions. Furthermore, because Aspire utilizes the standardized OpenTelemetry Protocol (OTLP), the telemetry configurations defined during development are directly extensible to production-grade collectors, ensuring architectural consistency across the entire lifecycle.

Q3: Bridging the Skills Gap for Small Teams

Aspire's unified .NET stack effectively bridged the skills gap for the I-Tech team, which lacked dedicated DevOps expertise. By abstracting the complexities of container orchestration and providing a familiar C#-based interface, Aspire enabled the team to manage a sophisticated distributed architecture using their existing professional competencies.

The availability of "Golden Paths"—pre-configured defaults for health checks, resilience (Polly), and service defaults—mitigated the risk of architectural misconfiguration. This confirms that high-level abstractions like Aspire can democratize access to cloud-native patterns for smaller engineering teams, allowing them to focus on delivering business value through the CRM system rather than being hindered by the steep learning curve of the broader cloud-native landscape.

Chapter 7

Conclusion

This thesis has explored the transformative potential of Aspire in bridging the gap between monolithic simplicity and microservices scalability. Through the practical implementation of the I-Tech CRM, we have demonstrated that the primary barrier to cloud-native adoption for SMEs is often not the complexity of the application logic itself, but the "infrastructure tax" associated with orchestration.

Aspire represents a significant evolution in this space. By abstracting service discovery, configuration management, and observability into a type-safe, C#-based orchestration layer, this tool allows developers to treat infrastructure as a direct extension of application code. This study confirms that a unified development stack—pairing Aspire with ASP.NET and Blazor—minimizes cognitive load and accelerates the "Inner Loop" development cycle, enabling small teams to manage complex distributed systems that would traditionally require dedicated DevOps resources.

7.1 Aspire Strengths

The introduction of Aspire represents a paradigm shift for software houses like I-Tech S.r.l. By abstracting the intrinsic complexities of Kubernetes and container orchestration, Aspire democratizes microservices architecture, allowing teams to adopt sophisticated patterns without prohibitive overhead.
The primary strengths of Aspire can be synthesized into the following domains:

- **Enhanced Developer Experience (DX):** Aspire prioritizes a unified development lifecycle. By providing a local "Inner Loop" that mirrors production-grade orchestration, it bridges the gap between coding and deployment.

- **Type-Safe Orchestration and Reliability:** Utilizing C# as a *Domain Spe-*

cific Language (DSL) ensures compile-time validation of service dependencies, drastically reducing runtime errors caused by configuration drift.

- **Native Observability and Diagnostics:** By embedding OpenTelemetry at its core, Aspire provides "zero-configuration" visibility, reducing the Mean Time to Identify (MTTI) for distributed failures.
- **Architectural Standardization:** Aspire's "opinionated" nature enforces industry best practices regarding resilience and security, ensuring the system is maintainable by default.
- **Open-Source Ecosystem and "Golden Paths":** Aspire standardizes proven architectural patterns for complex requirements, accelerating adoption and alignment with industry standards.
- **Longevity and Multi-Cloud Extensibility:** As a flagship .NET project, it mitigates software abandonment risk and allows for future expansion preventing vendor lock-in.

7.2 Limitations

While Aspire has proven to be a highly effective "Force Multiplier", several limitations must be acknowledged:

- **Ecosystem Dependency:** It remains a .NET-centric tool, which may limit its adoption in heterogeneous environments where non-Microsoft stacks are dominant.
- **Environment Fidelity:** Aspire's local orchestration cannot perfectly replicate production-grade variables such as geographical network latency or the nuances of managed third-party PaaS offerings.
- **Abstraction Risk:** By hiding the complexity of underlying manifests, there is a risk that developers may lack the fundamental knowledge required to troubleshoot low-level infrastructure failures.
- **Integration Maturity:** Several hosting integrations (notably for on-premise Kubernetes) are currently in preview releases, organizations must account for the "version risk" associated with non-GA features.

7.3 Future Work

7.3.1 Aspire Development

On the Aspire front, future iterations could focus on the automated generation of manifests for major cloud providers such as AWS (ECS/Fargate) or Google Cloud (Cloud Run). Standardizing the deployment pipeline across these environments would further validate the cloud-agnostic nature of Aspire. Furthermore, the stabilization of Docker and Kubernetes deployment integrations remains a significant milestone for Aspire's versatility in on-premise environments.

Additionally, future work could explore broader polyglot support. By leveraging Aspire's ability to generate language-agnostic manifests, the ecosystem could better accommodate heterogeneous teams, allowing them to benefit from standardized orchestration while utilizing diverse technology stacks.

7.3.2 Evolution of the I-Tech CRM

The CRM developed during this project now serves as a central strategic asset for I-Tech S.r.l. At the time of writing, the project is in its primary implementation phase, with the core architectural pillars and essential domain services successfully established. The immediate roadmap focuses on expanding the feature set to include the full suite of functionalities required by the sales and support departments, ensuring the system evolves into a comprehensive enterprise solution that fully replaces the legacy infrastructure.

The modular design inherent in this architecture facilitates rapid iteration; new services can now be developed independently, unencumbered by the technical debt of the previous monolith. By offloading the complexities of service orchestration to Aspire, I-Tech is now positioned to respond to market demands with significantly increased technical agility.

7.4 Final Remarks

The transition from a legacy PHP monolith to an Aspire-orchestrated system has been a paradigm shift for I-Tech S.r.l. While tools like Aspire represent a significant leap in accessibility, it is vital to remember that abstraction must not become a "black box". The power of these tools should be used as a springboard for learning, not a substitute for understanding. By maintaining a core comprehension of

networking and containerization, developers ensure they can navigate complex production environments. Ultimately, Aspire proves that with the right level of abstraction, small teams can build and scale modern software with the same sophistication as large-scale enterprise organizations.

List of Figures

2.1 Monolithic Architecture Diagram [42] 8
2.2 Microservices Architecture Diagram [43] 9

3.1 System Architecture Overview 29
3.2 Aspire Service Discovery via Environment Variable Injection [28] . 31
3.3 Aspire Resource Discovery Example 31
3.4 Aspire Resource Health and Readiness Overview 32
3.5 Aspire Traces Overview 35
3.6 Aspire Resource Components Overview 38

5.1 Initial architecture of the I-Tech CRM 50
5.2 Adding Aspire Orchestration to an existing Visual Studio solution . 52
5.3 Aspire Dashboard: Distributed trace showing a successful OIDC handshake and authorized request flow across services. 59

Bibliography

[1] Apache Software Foundation. *Apache HTTP Server Official Website*. 2026. URL: https://httpd.apache.org/ (visited on 2026-01-14).

[2] Cloud Native Computing Foundation. *Kubernetes Official Documentation*. 2026. URL: https://kubernetes.io/docs/ (visited on 2026-01-08).

[3] Docker Inc. *Docker Official Documentation*. 2026. URL: https://docs.docker.com/ (visited on 2026-01-08).

[4] Domain-Driven Design .NET. *Swashbuckle.AspNetCore GitHub Repository*. 2026. URL: https://github.com/domaindrivendev/Swashbuckle.AspNetCore (visited on 2026-01-14).

[5] Prof. Pietro Ferrara. *Lecture Notes of Software Architectures*. 2024. URL: https://www.unive.it/data/course/576809 (visited on 2024-08-15).

[6] FluentValidation. *FluentValidation Official Documentation*. 2026. URL: https://docs.fluentvalidation.net/ (visited on 2026-01-26).

[7] I-Tech S.r.l. *I-Tech S.r.l. Official Website*. 2026. URL: https://itechsoftware.it/ (visited on 2026-01-08).

[8] Meta. *React Official Website*. 2026. URL: https://react.dev/ (visited on 2026-01-26).

[9] Microsoft. *.NET Official Documentation*. 2026. URL: https://learn.microsoft.com/en-us/dotnet/ (visited on 2026-01-08).

[10] Microsoft. *ASP.NET APIs Documentation*. 2026. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/apis (visited on 2026-01-08).

[11] Microsoft. *ASP.NET Core Official Documentation*. 2026. URL: https://learn.microsoft.com/en-us/aspnet/core/ (visited on 2026-01-08).

[12] Microsoft. *ASP.NET Core Official Website*. 2026. URL: https://dotnet.microsoft.com/en-us/apps/aspnet (visited on 2026-01-26).

[13] Microsoft. *Aspire 13.0 Release Notes*. 2025. URL: https://aspire.dev/whats-new/aspire-13/ (visited on 2026-01-26).

[14] Microsoft. *Aspire Integration*. 2026. URL: https://aspire.dev/integrations/gallery/ (visited on 2026-01-08).

[15] Microsoft. *Aspire Official Documentation*. 2026. URL: https://aspire.dev/docs/ (visited on 2026-01-08).

[16] Microsoft. *Aspire Official Website*. 2026. URL: https://aspire.dev/ (visited on 2026-01-08).

[17] Microsoft. *Aspire Samples GitHub Repository*. 2026. URL: https://github.com/dotnet/aspire-samples (visited on 2026-01-26).

[18] Microsoft. *Aspire SQL Server Integration*. 2026. URL: https://aspire.dev/integrations/databases/sql-server/ (visited on 2026-01-08).

[19] Microsoft. *Blazor Official Documentation*. 2026. URL: https://learn.microsoft.com/en-us/aspnet/core/blazor/ (visited on 2026-01-08).

[20] Microsoft. *Blazor Official Website*. 2026. URL: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor (visited on 2026-01-26).

[21] Microsoft. *Entity Framework Core Official Documentation*. 2026. URL: https://learn.microsoft.com/en-us/ef/core/ (visited on 2026-01-08).

[22] Microsoft. *eShop on Aspire GitHub Repository*. 2026. URL: https://github.com/dotnet/eShop (visited on 2026-01-26).

[23] Microsoft. *Kestrel Web Server Documentation*. 2026. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel (visited on 2026-01-26).

[24] Microsoft. *LINQ (Language-Integrated Query) Documentation*. 2026. URL: https://learn.microsoft.com/en-us/dotnet/csharp/linq/ (visited on 2026-01-14).

[25] Microsoft. *MAUI Official Website*. 2026.
URL: https://dotnet.microsoft.com/en-us/apps/maui (visited on 2026-01-26).

[26] Microsoft. *Visual Studio Official Website*. 2026.
URL: https://visualstudio.microsoft.com/ (visited on 2026-01-08).

[27] Microsoft. *Xamarin Official Website*. 2026.
URL: https://dotnet.microsoft.com/en-us/apps/xamarin (visited on 2026-01-26).

[28] Milan Jovanovic. *How .NET Aspire Simplifies Service Discovery*. 2025.
URL: https://www.milanjovanovic.tech/blog/how-dotnet-aspire-simplifies-service-discovery (visited on 2026-01-26).

[29] OpenID Foundation. *OpenID Official Website*. 2026.
URL: https://openid.net/ (visited on 2026-01-08).

[30] OpenIddict. *OpenIddict Official Documentation*. 2026. URL: https://documentation.openiddict.com/ (visited on 2026-01-08).

[31] OpenTelemetry. *OpenTelemetry Official Website*. 2026.
URL: https://opentelemetry.io/ (visited on 2026-01-08).

[32] Polly. *Polly Official Documentation*. 2026.
URL: https://www.pollydocs.org/ (visited on 2026-01-26).

[33] PostgreSQL Global Development Group. *PostgreSQL Official Documentation*. 2026.
URL: https://www.postgresql.org/docs/ (visited on 2026-01-08).

[34] RabbitMQ. *RabbitMQ Official Website*. 2026.
URL: https://www.rabbitmq.com/ (visited on 2026-01-14).

[35] M. Richards and N. Ford. *Fundamentals of Software Architecture: A Modern Engineering Approach*. O'Reilly Media, 2025.
URL: https://books.google.it/books?id=yCl0EQAAQBAJ.

[36] Swagger. *Swagger Official Website*. 2026.
URL: https://swagger.io/ (visited on 2026-01-14).

[37] TabBlazor. *TabBlazor GitHub Repository*. 2026. URL: https://github.com/TabBlazor/TabBlazor (visited on 2026-01-08).

[38] TabBlazor. *TabBlazor Official Website*. 2026.
URL: https://tabblazor.com/ (visited on 2026-01-08).

[39] Tabler. *Tabler GitHub Repository*. 2026.
URL: https://github.com/tabler/tabler (visited on 2026-01-08).

[40] Tabler. *Tabler Official Website*. 2026.
URL: https://tabler.io/ (visited on 2026-01-08).

[41] The Twelve-Factor App. *The Twelve-Factor App Official Website*. 2026.
URL: https://12factor.net/ (visited on 2026-01-08).

[42] Web and Crafts. *Understanding Monolithic Architecture: Pros and Cons*. 2024.
URL: https://webandcrafts.com/blog/monolithic-architecture (visited on 2026-01-26).

[43] Web and Crafts. *What is Microservices Architecture? Advantages and Disadvantages*. 2024.
URL: https://webandcrafts.com/blog/what-is-microservices-architecture (visited on 2026-01-26).

[44] Wiki.js. *Wiki.js Official Website*. 2026.
URL: https://js.wiki/ (visited on 2026-01-26).

[45] Zucchetti spa. *Mago Official Website*. 2026.
URL: https://www.mago-erp.com (visited on 2026-01-08).

[46] Zucchetti spa. *TaskBuilder Framework Official Website*. 2026.
URL: https://www.mago-erp.com/technology-and-development/platform-overview/taskbuilder-framework (visited on 2026-01-26).

[47] Zucchetti spa. *Zucchetti Official Website*. 2026.
URL: https://www.zucchetti.it/ (visited on 2026-01-08).