Software Architecture Documentation for Digital-Gemeinde.ch

Contents

[Version 1](#_Toc1605648693)

[1. Introduction 1](#_Toc729738005)

[2. Context and Constraints 2](#_Toc57721045)

[2.1 Business Context 2](#_Toc33421947)

[2.2 Technical Context 2](#_Toc1443692357)

[2.3 Constraints 3](#_Toc963522883)

[3. Architecture Overview 3](#_Toc697983960)

[3.1 Logical Architecture 4](#_Toc250299065)

[4 Detailed View 5](#_Toc1925428323)

[4.4 Storage 11](#_Toc183153967)

[4.5 Security Framework 12](#_Toc975996020)

[4.5 Infrastructure 12](#_Toc1058249885)

[5 Dataflows 13](#_Toc2115569798)

[5.1 Form Creation and Publishing 13](#_Toc1125649990)

[5.2 Form Display and Access by End User 14](#_Toc469479501)

[5.3. Form Submission 14](#_Toc842981393)

[6 Deployment Architecture 15](#_Toc1992480682)

[6.1 Deployment Architecture (Generic) 15](#_Toc270361711)

[6.2 Cloud Deployment (Azure) 17](#_Toc749415144)

[6.3 On-Premise or Private Cloud Deployment 17](#_Toc1011545075)

[6.2 CI/CD 17](#_Toc267694073)

[6.3 Network and Security 18](#_Toc453366926)

[6.4 Scalability 18](#_Toc620042599)

[7 Maintenance and Evolution 19](#_Toc359281534)

[7.1 Maintainability 19](#_Toc1397912552)

[7.2 Upgrades and Roadmap 19](#_Toc1701114011)

[7.2.1 Short-Term Initiatives (Next 3–6 Months) 19](#_Toc1218821052)

[7.2.2 Mid-Term Initiatives (6–12 Months) 20](#_Toc1400566112)

[7.2.3 Long-Term Initiatives (12+ Months) 20](#_Toc1080583624)

[Appendix 21](#_Toc2082313478)

## Version

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Description | Author |
| 1.0 | 25/06/2025 | Initial Version | Roger Vial |
|  |  |  |  |

## 1. Introduction

This document outlines the software architecture of the Digital-Gemeinde.ch solution developed by CumulusPro. It provides detailed insights into its components, design principles, deployment strategies, and the rationale behind technical decisions.

The target audience includes software architects, developers, DevOps engineers, IT administrators, and customer stakeholders involved in deployment and governance.

## 2. Context and Constraints

### 2.1 Business Context

Digital-Gemeinde.ch is a digital form management solution developed for the public sector. It enables authorities and institutions to create, manage, and deploy digital forms used in citizen services, internal processes, and administrative workflows.

The platform supports integration with existing government systems, including portals such as the iGovPortal, and aligns with common interoperability and identity standards used in the public administration domain.

It is suitable for deployment in public cloud, private cloud, or on-premise environments, allowing organizations to choose the infrastructure that fits their operational, legal, and data protection requirements.

### 2.2 Technical Context

The solution is designed as a modular, service-based architecture that can be deployed in cloud, hybrid, or on-premise environments. It supports both public and private infrastructure and integrates with local or cloud-based services depending on the operational context.

The application follows a clear separation of concerns across the frontend, business logic, and persistence layers, enabling independent deployment, scaling, and maintenance of each component.

It uses standard HTTP APIs, supports multi-tenancy, and persists structured data in relational databases, with unstructured content (such as form definitions and submissions) stored in external file storage systems. These components can be mapped to services available in Microsoft Azure, self-hosted infrastructure, or other supported platforms.

Authentication and identity management are integrated via standards-compliant protocols (OAuth 2.0 / OpenID Connect) and can connect to external identity providers such as Microsoft Entra, national identity services, or local systems.

Workflow automation is implemented through a loosely coupled interface that supports integration with external workflow engines, such as CumulusPro Straatos, or customer-specific solutions.

Notification services for email or system events can be connected through pluggable adapters, including commercial services or on-prem mail servers.

The solution is developed with an open-source approach. Core components and shared libraries use open standards and are packaged to allow extension, reuse, and transparent governance.

### 2.3 Constraints

The architecture must support deployment models that comply with Swiss legal requirements, including data residency and operational jurisdiction.

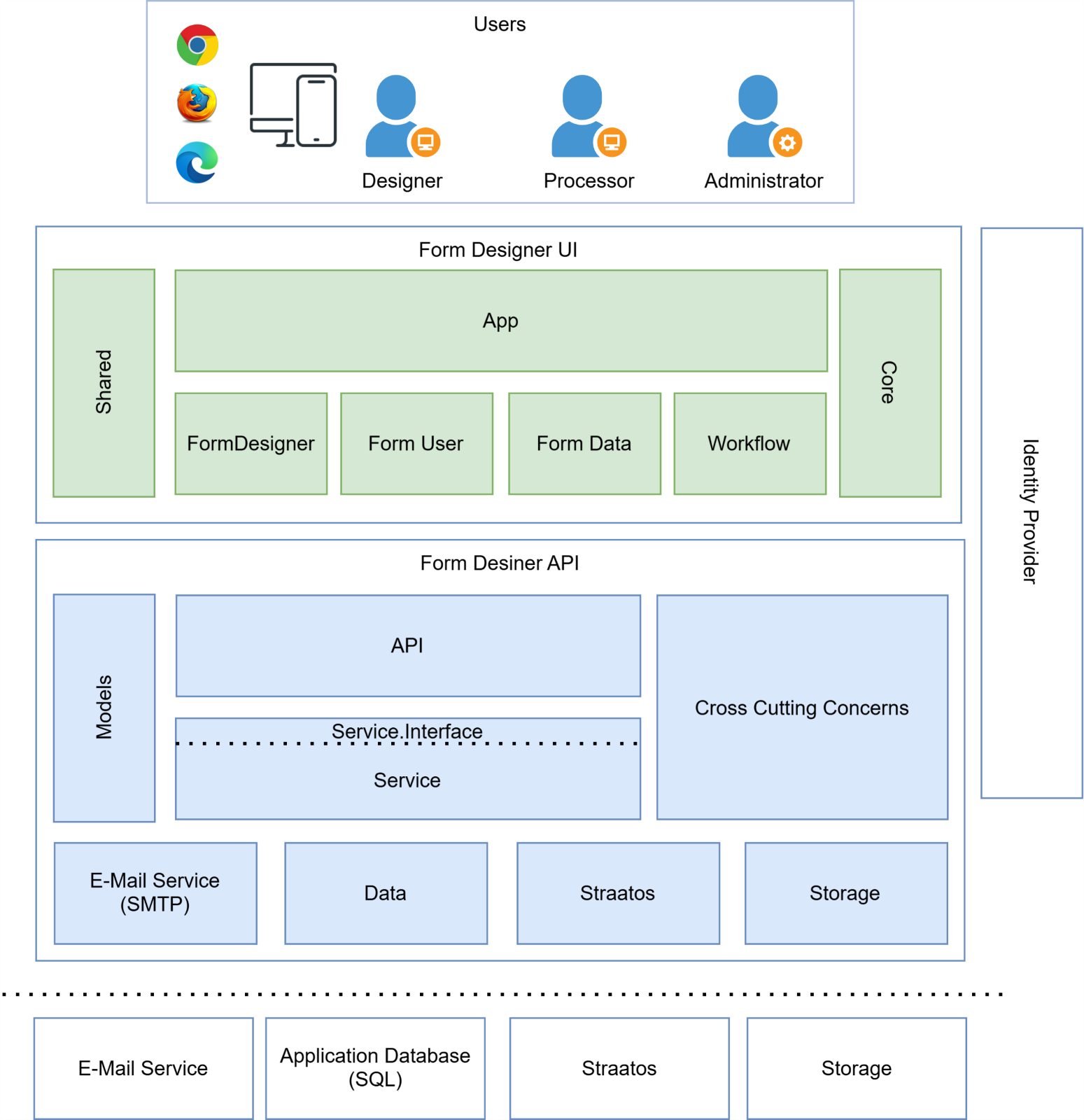
Multi-language support is an architectural consideration. The user interface must accommodate multiple languages, requiring the frontend to structured to allow language-specific content and labels to be managed externally.

The system must be maintainable over a multi-year lifecycle. This impacts technology choices, modular design, and update mechanisms. Components must be loosely coupled to allow for targeted updates and long-term support without disrupting core services.

## 3. Architecture Overview

### 3.1 Logical Architecture

The Digital-Gemeinde.ch solution is based on a layered modular architecture, which separates concerns across the presentation, business, and persistence layers.



The presentation layer is a Single Page Application (SPA) developed in Angular.

The business layer (Digital-Gemeinde.ch Backend) exposes RESTful APIs developed using .NET Core, implementing the core logic of form handling, validation, and state management.

The persistence layer is composed of SQL for relational data and either Azure Blob Storage or a local network storage for unstructured form definitions and submissions.

Integration is achieved through clearly defined interfaces that allow communication with external systems, such as Straatos for workflow management, Azure B2C for identity services, and SendGrid for email delivery.

### 4 Detailed View

The system is divided into several functional components:

The Form Designer Module enables users to create and manage form structures, including support for versioning.

The Form Submission Module provides interfaces for citizens and administrative users to complete and submit forms.

The Form Processor Module is responsible for managing the routing and processing of form submissions via external workflows.

The Admin Module offers functionality for managing users, roles, and tenant-specific configurations.

The Notification Module handles outbound communications such as submission confirmations or escalation alerts.

Lastly, the Audit and History Module maintains a historical log of all relevant activities and changes for traceability and compliance purposes.  
  
**4.1 Digital-Gemeinde.ch Designer Frontend**

The Digital-Gemeinde.ch Designer Frontend component is the core interface for creating and managing digital forms. It provides a form builder powered by Angular, allowing administrative users to define form elements, structure, and validation rules. This component supports the creation of multiple versions of each form and automatically generates a JSON schema that serves as the backend representation. Additional functionalities include tagging and search features to find existing forms.

The Form User module is designed for the end-user experience. It renders live forms based on the JSON schema and adapts to both desktop and mobile interfaces. The module performs client-side validation to ensure that all mandatory fields and formatting rules are met before submission. On submission, it securely transmits the user’s data to the backend via RESTful API calls. It also includes user feedback mechanisms, such as submission confirmations and error prompts.

The Workflow Panel is embedded within the user interface and either provides status updates within the Form designer or allows to connect directly to the Straatos workflow engine. This component allows users—particularly internal administrative staff—to view, process, and update workflow-related tasks without leaving the Digital-Gemeinde.ch interface. It supports task metadata retrieval, form state visualization, and contextual actions depending on the workflow status.

The Core and Shared modules serve as foundational building blocks across the Angular application. These modules encapsulate reusable services, including API clients for communication with the backend, HTTP interceptors for authentication and error handling, and route guards for access control. Shared utilities and models are also housed here, promoting consistency and reducing code duplication across the application.

Technologies:

* Angular, TypeScript, HTML/CSS

Key Features:

- Form builder interface

- API interaction for saving and retrieving form data

- Dynamic rendering based on JSON schema

Subcomponents:

* Form Designer – Core component responsible for building and editing form structures. F
* Form User – Creating and managing user access.
* Form Data – Handles display and management of submitted data for users.
* Workflow – Manages the UI logic related to form status transitions (e.g., Draft, Submitted).
* Core – Provides shared logic, configuration, and utilities used across frontend modules.
* Shared – Contains reusable UI elements (buttons, modals, inputs), common styles, and helper functions.

Interactions:

* Retrieves/stores JSON form schemas from Storage via backend

**4.2 Digital-Gemeinde.ch Designer Backend**

The API Controllers serve as the entry points for frontend interactions. Each controller corresponds to a distinct business domain, such as form design, form data submissions, or user management. These controllers receive HTTP requests, validate input parameters, and delegate processing to the appropriate service components. They follow RESTful conventions and support versioning, authentication, and structured error handling to ensure consistency and reliability across the API surface.

The Service Layer is responsible for encapsulating the core business logic of the application. It handles form creation, data validation, submission tracking, and workflow orchestration. Services ensure that business rules are applied consistently and act as intermediaries between the API layer and data access logic. By organizing logic into distinct service classes, the architecture maintains separation of concerns and supports modular development and testing.

The Data Layer interacts directly with the Azure SQL database using Entity Framework Core. It includes entity configurations, context definitions, and repository patterns that abstract low-level SQL operations. This layer also includes mapping logic between domain models and Data Transfer Objects (DTOs), ensuring a clear distinction between internal structures and external-facing responses.

The Straatos Integration module provides an abstraction for communicating with the external Straatos workflow platform. It encapsulates logic for initiating workflows, retrieving task states, and sending form metadata to be processed in task queues. This module is designed to support extensibility and handles authentication, retry logic, and exception management when interfacing with external services.

The Storage Adapter manages all interactions with Azure Blob Storage. It provides functions to upload, retrieve, and delete JSON-based form schemas, as well as attachments and completed documents. It ensures secure access through signed URLs and manages file lifecycle operations in conjunction with form submission states.

The Email Adapter encapsulates email-related operations. It formats messages based on configurable templates and dispatches them to recipients using either a local E-Mail Service via SMTP or the SendGrid API. This adapter is used for sending notifications about form submission confirmations, assignment notifications, and escalation alerts. It includes mechanisms for logging, retrying failed sends, and handling delivery responses.

Technologies:

* ASP.NET Core, Entity Framework Core

Key Services:

* FormDesignService
* SubmissionService
* ProcessorService

Subcomponents:

* Model – Defines data structures and domain entities used throughout the backend.
* Cross Cutting Concerns – Handles logging, validation, error handling, and security aspects that are reused across services.
* API – Exposes REST endpoints to frontend components via controllers and API contracts.
* Service – Contains the core business logic for handling forms, submissions, users, and state transitions.
* E-Mail Service / SMTP – Provides integration with E-Mail services for sending notification and confirmation emails.
* Data – Responsible for database access using repositories and Entity Framework context.
* Storage – Encapsulates logic to interact with blob storage for uploading and retrieving JSON and document files.
* Controllers – Expose RESTful APIs for form operations (e.g. `FormController`, `SubmissionController`).
* Services – Contain business logic for each domain (e.g. `FormDesignService`, `UserService`).

Interactions:

* Reads/writes to Database
* Uploads/downloads JSON and documents from Storage
* Serves APIs consumed by Digital-Gemeinde.ch Designer Frontend

**4.3 SQL Database**

The FormDesigns table contains the active definitions of form templates. Each entry includes metadata such as name, tenant ID, creator, version, and timestamps for creation and updates. It links to JSON schema files stored in Azure Blob Storage, enabling dynamic rendering of forms in the frontend.

The FormDesignsHistory table stores previous versions of each form template, enabling rollback and audit functionality. This table maintains a historical snapshot of metadata and links to archived schema files, supporting traceability and compliance.

The FormDatas table holds all submitted form records. It captures the form ID, tenant information, user-submitted values, and links to any files or documents stored externally. It is used as the main source for querying submitted data and supports indexing for fast lookup and reporting.

The Users table stores metadata about system users. It includes their unique identifier from Azure AD B2C, contact details, assigned roles, and audit trail information like creation and update timestamps. This table supports administrative functions such as permission management and user lifecycle tracking.

The Processor and Designer tables link users to specific forms and define their respective roles in processing and authoring. These associations are critical for enforcing role-based access control (RBAC) and for filtering dashboards and access scopes by user type.

The FormStatesConfig table defines the permissible states and transitions for each form. It stores label-value mappings and references to the related form design. It is used during workflow execution to enforce allowed transitions and determine the next possible states in user interfaces.

Key Tables:

* FormDesigns / FormDesignsHistory
* FormDatas
* FormTemplates
* FormStatesConfig / FormStatesConfigHistory
* Designer / DesignerHistory
* Processor / ProcessorHistory
* Users

Responsibilities:

* Ensures transactional consistency and auditability
* Enforces foreign key relationships for version and tenant integrity

### 4.4 Storage

Stores large unstructured data including:

* JSON form design definitions
* Submitted form payloads
* Uploaded templates and documents

Subcomponents:

* FormDesign JSON Files – Defines the layout and logic of forms.
* Submitted Data Files – Persisted submissions in raw format.
* Template Files – DOCX/HTML templates for PDF generation or export.

Access:

* Storage URLs are referenced from SQL tables
* Accessed by backend for read/write operations

### 4.5 Security Framework

Role-Based Access Control (RBAC) is enforced throughout the application to limit access based on user roles. Each user is assigned a role such as administrator, processor, or viewer, and these roles dictate the permitted actions within the application. The RBAC logic is applied on both frontend routing and backend API endpoints.

Identity provider can either be a local Identity Provider (IdP),Azure Entra or Azure AD B2C that handles the identity and access management. The system uses OAuth 2.0 and OpenID Connect protocols for secure token-based authentication. All access tokens are validated server-side for every request. JWT Token Validation ensures that only authenticated users can access secured APIs.

Encryption is implemented both in transit and at rest. HTTPS is enforced across all endpoints to protect data in motion. SQL and File Storage can be configured to use server-side encryption to secure data at rest.

Audit Logging is implemented across all critical operations. The system logs changes to form definitions, user actions, and workflow events. These logs are stored securely and can be exported for compliance reviews and forensic analysis.

Interactions:

* Used by Digital-Gemeinde.ch Designer Frontend to authenticate users
* Tokens are passed to Digital-Gemeinde.ch Designer Backend to enforce access control

### 4.5 Infrastructure

Provides the hosting, scaling, and deployment environment for the full Digital-Gemeinde.ch Designer solution.

This is either provided by CumulusPro when a hosted model is used or by the partner/enduser for on Premise Solutions.

Responsibilities:

* Ensures high availability and secure deployment
* Supports automated deployment, rollback, and monitoring

### 5 Dataflows

### 5.1 Form Creation and Publishing

Actors: Administrator or Designer

A diagram of a flowchart

AI-generated content may be incorrect.   
Flow:

1. User accesses the Digital-Gemeinde.ch Designer UI via the frontend.
2. Form schema and configuration are created in the browser using the form editor.
3. A POST /api/formdesigns request is sent to the backend.
4. Backend validates and persists the design in the SQL Database (FormDesigns table) and Storage (Azure BLOB Storage or Network Share)
5. Optional: The form is marked IsActive = true and becomes available for public access.

Notes:

* Data is versioned and linked to the TenantId.
* All changes are tracked in the FormDesignsHistory table.

Shape

### 5.2 Form Display and Access by End User

Actors: Public User via Kantons or Gemeinde Portal

A screenshot of a computer screen

AI-generated content may be incorrect.  
Flow:

1. The Portal embeds the Digital-Gemeinde.ch Designer frontend using an IFrame or dynamic load.
2. The form ID is passed via URL or configuration.
3. A GET /api/formdesigns/{id} request loads the form schema.
4. The form is rendered dynamically in the browser using JSON metadata.

Notes:

* No login is required if the form is public.
* Portals may prefill form fields via query parameters.

Shape

### 5.3 Form Submission

Actors: End User

A diagram of a network

AI-generated content may be incorrect.  
Flow:

1. User completes the form in the frontend.
2. On submission, a POST /api/formdatas request is sent.
3. Backend validates input and writes to the FormDatas table.
4. The submitted document is uploaded to Storage.
5. Webhooks are triggered (e.g., to DMS or notification services).
6. An email confirmation is sent if enabled.

Notes:

* The form submission includes metadata, status, and timestamps.
* Traceability is maintained through the Id and SubmittedDate.

Shape

Shape

## 6 Deployment Architecture

The solution supports both cloud-based and on-premise deployments. Customers can choose between hosting the platform in Microsoft Azure for managed scalability, or operating it within their own data centers or private cloud environments for greater control over infrastructure and compliance. Both models share the same core architecture and functionality, with infrastructure components adapted to the respective environment.

### 6.1 Deployment Architecture (Generic)

A diagram of a cloud

AI-generated content may be incorrect.

The two green rectangles represent the **Digital-Gemeinde.ch components**. All other components shown in the diagram are provided either by the customer’s infrastructure or, in the case of cloud deployments, by the CumulusPro-managed hosting environment.

**DMS/Secure Zone**: Both Digital-Gemeinde.ch components must be deployed in a network zone that allows inbound internet access, enabling users to load the frontend and interact with backend APIs.

The **Digital-Gemeinde.ch Frontend** is a static web application. It is delivered to users via the browser and executes entirely on the client side. All data interactions and API requests are routed through the **Digital-Gemeinde.ch Backend component**.

While the frontend and backend can be hosted on the same web server, they are shown as separate components in the diagram to clarify their distinct roles and communication paths.

**API Management** is optional but recommended if a more fine-grained control over exposed endpoints and rate limiting is required.

### 6.2 Cloud Deployment (Azure)

When deployed to the Microsoft Azure Cloud, both Digital-Gemeinde.ch Frontend and Backend can be deployed in an Azure App Service. This allows for scaling by providing multiple instances of the service.

Data is stored in Azure SQL Server and can be scaled dynamically. JSON files are stored in Azure Storage Accounts (BLOB Storage).

Monitoring and Log Analytics are provided by Application Insights and Log Analytics.

### 6.3 On-Premise or Private Cloud Deployment

For customers requiring deployment in their own data center or private cloud, the solution can be installed on virtual machines or container orchestration platforms such as Kubernetes.

The frontend is delivered as a static Angular application hosted by a local web server (e.g., NGINX or IIS), and the backend runs on Linux or Windows servers with .NET Core runtime.

SQL Server is used as the database engine, and form assets are stored on a local or network file system.

Authentication can be integrated with customer-provided identity providers supporting OAuth2 and OpenID Connect.

Logging and monitoring integrate with local tools or can be configured for external observability platforms depending on policy.

### 6.2 CI/CD

The CI/CD pipeline is implemented using Azure DevOps. Source code is version-controlled in Azure Repos and is automatically built and tested upon each commit.

The pipeline includes distinct stages for building the Angular frontend and .NET Core backend, executing automated unit and publishing build artifacts.

Releases are orchestrated using multistage deployment pipelines that promote artifacts through QA, staging, and production environments.

Customers using their own CI/CD can download the artifacts or source code from CumulusPro and use their own CI/CD infrastructure for deployments.

### 6.3 Network and Security

All traffic from the browser to the Digital-Gemeinde.ch components is via HTTPS with TLS 1.2 or higher.

Firewalls/WAFs should be configured in front of the application service. It is also recommended to have an API Management before the Digital-Gemeinde.ch Backend Component.

CORS (Cross-Origin Resource Sharing) policies should be configured in the backend to restrict requests to known frontend origins, enhancing defense against unauthorized access.

### 6.4 Scalability

The solution is designed to scale efficiently in both cloud and on-premise environments. In cloud deployments, scalability is managed through Azure App Services and elastic database pools. The frontend and backend services can be scaled independently based on load, and the platform supports horizontal scaling through App Service Plan instances.

For on-premise installations, the system has a low baseline resource footprint and does not require extensive infrastructure to operate effectively. Scalability is achieved primarily by increasing the capacity of the web server hosting the frontend and by deploying additional instances of the backend service across multiple servers. The backend is stateless and can be replicated.

### 6.5 Scalability Example Calculation

**Context:** Digital-Gemeinde.ch is hosted on a Windows Server with IIS. Each time a user opens the application, approximately 5MB of static content is downloaded by the browser. This is a higher limit and assumes some graphic elements such as a logo is loaded too. The application then makes two API calls:

* One 50KB GET request to retrieve a form definition (JSON file)
* One 50KB POST request to submit the form

Each complete form interaction (1 user session) thus includes approximately 5.1MB of total data transferred.

**Assumptions**

* Server has a 1 Gbps NIC (gigabit network interface card)
* IIS is properly configured with asynchronous request handling, output caching, and no unnecessary logging overhead
* Disk I/O and CPU are not the bottlenecks
* Each user performs one form submission per session
* The application is stateless and sessions are not persisted in server memory

**Calculations**

*Per User Data Transfer*

* Static load: 5MB
* API form GET + POST: 2 × 50KB = 100KB
* Total per user: **5.1MB**

*Network Constraint (1 Gbps Effective Bandwidth)*

* Effective throughput: ~120MB/sec
* Max users/sec: 120MB / 5.1MB ≈ 23 users/sec
* Max users/hour: 23 × 3600 ≈ **82,800 users/hour**

*CPU/IIS Constraint*

* With optimized async handling, IIS can process 500–2000 concurrent requests
* Assuming minimal processing per API request, concurrent form interactions supported: **~1000–2000 users**

**Scaling with Multiple IIS Servers** If horizontal scaling is applied (each server with its own 1 Gbps connection):

|  |  |
| --- | --- |
| **IIS Servers** | **Estimated Max Forms/hour** |
| 1 | ~82,800 |
| 4 | ~331,000 |
| 10 | ~828,000 |

**Other Optimization Options**

* Use a CDN (e.g., Azure Front Door, Cloudflare) to cache and deliver the 5MB static bundle closer to users

**Conclusion** The primary bottleneck in this scenario is network bandwidth due to the 5MB static payload per new user session. With a 1 Gbps network, ~82,800 unique users can be supported per hour. This can be increased linearly with additional IIS servers and optimized further with a CDN and front-end performance tuning.

## 7 Maintenance and Evolution

### 7.1 Maintainability

The solution follows modular design principles to enhance maintainability. Code for both frontend and backend is organized by feature domain, with clearly defined interfaces and services.

This structure enables teams to isolate changes and implement new functionality without affecting unrelated modules. Each service and component include inline documentation and comments, promoting code readability and maintainability.

The use of shared libraries and reusable components ensures consistency and reduces duplication across the codebase.

### 7.2 Upgrades and Roadmap

The technical roadmap outlines planned upgrades and architecture improvements to ensure long-term sustainability, scalability, and maintainability of the Digital-Gemeinde.ch  platform.

### 7.2.1 Short-Term Initiatives (Next 3–6 Months)

* Frontend Framework Alignment and Upgrade
  + Migrate all components to Angular 18 (currently a mixed use of Angular 16, 17, and 18 packages)
  + Remove legacy dependencies (e.g. @angular-ru/\*) in favor of Angular-native tooling
* Backend Platform Consolidation
  + Ensure all backend projects target .NET 8.0 (currently a mix of net8.0 and netstandard2.1)
  + Improve Maintainability
  + Improve Libraries with low activity/freshness
* Infrastructure Improvements
  + Introduce CI/CD pipelines with code quality gates (SAST, dependency scanning)
  + Generate deployment templates for Container based deployments
* Observability and Monitoring
  + Standardize logging across backend modules
  + Enable distributed tracing using OpenTelemetry

### 7.2.2 Mid-Term Initiatives (6–12 Months)

* API Strategy Enhancements
  + Introduce API versioning and deprecation strategy
  + Generate and maintain Swagger/OpenAPI documentation for all endpoints
* Build and Security
  + Integrate automated vulnerability and license scans in CI pipelines
* Storage and Data Strategy
  + Introduce versioned form submission schema
  + Introduce search and retrieve capabilities for submitted form data from the backend
* Introduce Application level retention policies

### 7.2.3 Long-Term Initiatives (12+ Months)

* Architectural Evolution
* Performance and Scalability
* Introduce caching layer for frequently accessed form designs and templates
* Developer Experience
* Improve test coverage and introduce test data generation utilities

## Appendix