ChatGPT

Concepts and Overview

Project Background: Yapidoo (formerly *Yalldoo*) is a social networking platform designed to help people organize and join real-world events with friends or new acquaintances. Unlike traditional event apps focused on large public events, Yapidoo enables any individual to create an event and gather people, or to join events created by others [1]. For example, if two players drop out of a friendly football match, a user can create an event on Yapidoo to find replacements among friends or nearby users [2]. The platform's goal is to ensure no one misses out on activities due to lack of company, by making it easy to find events or companions anytime and anywhere [3].

Migration Rationale: To achieve this vision at scale, the Yapidoo service must be modernized with current technologies and best practices. The existing codebase (built on earlier versions of .NET and Angular) will be migrated to the latest **.NET** (for back-end) and **Angular** (for front-end) frameworks. This migration aims to improve performance, maintainability, and scalability of the platform. Upgrading to the latest **.NET** version provides better performance and cloud-native capabilities (async improvements, cross-platform support, etc.), while migrating the front-end to the latest **Angular** version will enhance user experience and leverage modern UI features. We will also re-architect the application into a **microservices** style and adopt cloud-native deployment (Docker containers orchestrated via Kubernetes) for reliability and scalability [4] [5]. Furthermore, we plan to integrate with Google Cloud services (Firebase, Google APIs) as needed to take advantage of ready-made solutions (for example, managed authentication, real-time databases, or mapping services) and to remain cloud-agnostic, considering alternatives to the previously favored Azure/AWS services. Overall, the migration will position Yapidoo on a solid technological foundation, enabling continuous delivery of new features and a smoother user experience.

Modernized Architecture Overview: The new Yapidoo architecture will consist of a **suite of microservices** in the back-end, a single-page Angular front-end, and robust DevOps pipelines for continuous integration and deployment. Each back-end microservice will handle a specific domain (for example, **User/Auth Service**, **Event Management Service**, **Social/Friend Service**, etc.), aligning with domain-driven design principles to keep services loosely coupled [6]. These .NET services will expose RESTful APIs (using ASP.NET Core Web API) and communicate with each other where necessary through well-defined interfaces (possibly using an API gateway or message bus for inter-service communication). Each service will manage its own data (database per service) to ensure **data independence** and avoid tight coupling [7]. The front-end will be an **Angular** application that consumes these APIs and provides a responsive, dynamic user interface following the designs (as per the Figma prototypes). The UI will support multiple languages and locales from the start, automatically choosing the site language based on the user's region with an option to switch language manually [8]. The platform will incorporate **social login options** (e.g. Google and Facebook authentication) to streamline user onboarding [9], and will leverage modern authentication protocols (OAuth2/OIDC with JWT tokens) for securing API calls.

Key concepts and improvements in this migration include:

- **Continuous Integration & Deployment (CI/CD):** We will set up automated CI/CD pipelines to build, test, and deploy both back-end and front-end components. Every code push will trigger automated builds and test suites, ensuring that new changes are validated. Upon successful tests, the pipeline will produce Docker images for each service and deploy them to the staging/

production Kubernetes cluster. This automation will drastically reduce manual effort and risk in deployments, enabling frequent and reliable releases [10] [11]. A tool such as GitHub Actions, Azure DevOps, or Jenkins will be used to implement the pipeline (for example, Jenkins was demonstrated to automate publishing of back-end and front-end Docker images to a K8s cluster [12]). The CI/CD process will also include steps for code quality checks, security scans, and artifact publishing.

- **Containerization & Kubernetes Deployment:** All application components will be containerized using **Docker** for consistency across environments. Docker images will be created for each microservice (using lightweight base images for .NET runtime) and for the Angular front-end (which can be served via an Nginx container or via a .NET static file server). Using containers ensures that the software runs the same way on development, test, and production, eliminating environment-specific issues. To manage these containers at scale, we will use **Kubernetes (K8s)** as the orchestration platform. Kubernetes will handle deployment, scaling, and resilience of the services automatically [4] [5]. For example, Kubernetes can monitor the .NET service instances and restart any that crash, and can scale out additional instances in response to load. We plan to use a managed Kubernetes service (such as Google Kubernetes Engine on GCP, or an equivalent on Azure/AWS) to reduce operational overhead [13]. Kubernetes will also simplify rolling updates and version management, as well as provide service discovery and load balancing between microservice pods. By migrating to Kubernetes, we harness its capabilities for improved scalability, reliability, and efficiency in our deployments [4].

- **Cloud Provider Integration (Google Cloud):** As an alternative to relying solely on Azure or AWS, we will integrate **Google Cloud Platform (GCP)** services where appropriate. This includes deploying our containerized apps on GCP's Kubernetes service (GKE) and potentially using **Google Firebase** for certain PaaS features. For instance, Firebase Authentication can be evaluated to offload user identity management (especially for social logins), Firebase Cloud Storage could be used for storing user-uploaded images, and Firebase Cloud Messaging for push notifications. We will also consider using Google's mapping APIs (Google Maps/Places API) for location-based features – e.g. providing an auto-complete for city or place names when creating an event, as hinted by the requirement to choose city from an auto-complete input [14]. The aim is to leverage PaaS offerings to speed up development (using well-tested components for common needs like auth, storage, messaging), while ensuring the solution remains portable. The architecture will remain cloud-agnostic to a degree – since we use Kubernetes and standard technologies, we could deploy on AWS or Azure similarly – but GCP will be a primary target environment for now. All secrets or keys for external APIs (Google API keys, etc.) will be managed securely (using something like Kubernetes Secrets or a service like Google Secret Manager) and not hard-coded.

- **Legacy Codebase Migration:** The existing code will be carefully migrated to the new structure. Rather than attempting an in-place upgrade, we will **create a new project repository** (or set of repositories) for the revamped platform. Core business logic from the old system can be refactored and ported over to the new .NET services if it is well-written, but largely we will **rewrite components** to fit the new architecture, taking advantage of improvements in .NET API design and Angular. The migration will follow an iterative approach: first rebuild the foundational pieces (e.g., the data models, entities, and key APIs) in the new stack, then gradually implement all required features to reach parity with the old system. We will set up the new solution structure to host multiple microservice projects and a front-end project, possibly adopting a **mono-repo** for easier coordination (with separate folders for each service and the UI). The database schema will also be updated as needed – each microservice may have its own database (following a database-per-service pattern for loose coupling [7]), and we'll plan data migration scripts to move data from the old schema to new ones if an existing database with user/event data exists. Until the new system is ready, the old system can continue running; we might plan for a **cut-over** deployment where we switch to the new system once it's fully tested and populated with necessary data. Risk is mitigated by extensive testing and possibly running the new system in parallel (beta mode) before full switch.

- **Multi-Language Interface & Content:** From day one, the new front-end will support **internationalization (i18n)** to cater to a diverse user base (for example, English and Ukrainian are needed given Yalldoo's origin). The application will detect the user's locale and default to the appropriate language, while allowing manual language selection [8]. In practice, this means using Angular's i18n capabilities or a library like **ngx-translate** to manage translation files for all UI text. We will mark all user-facing strings for translation and maintain language JSON files or use Angular's built-in xlf format for translations [15]. Initially, we can include the primary languages (e.g., English, Ukrainian, possibly others) and ensure the platform is easily extensible to more languages. The content that users create (event titles/descriptions) will remain in whatever language the user provides; however, the UI chrome (menus, buttons, messages) will be localized. We will also account for regional settings like date/time formats and possibly units within the app by utilizing Angular's localization features (built-in pipes for dates/numbers etc., which adapt to locale [16]). By building multi-language support in the core, we ensure a wider reach and a personalized experience in the user's native language from the landing page onward.

- **Image Storage & Processing:** Yapidoo involves user-uploaded images (for example, event photos, user profile pictures, or post-event "report" photos). Storing and serving these images efficiently is a key consideration. Instead of storing images on the application server's filesystem, we will use **cloud blob storage** (like Google Cloud Storage or an alternative) to store all user images. This provides durable, scalable storage and allows leveraging CDNs for delivery. For instance, images can be stored in a Google Cloud Storage bucket, and served via a **Content Delivery Network (CDN)** for fast access globally [17]. We will implement an image handling service or module that on image upload will perform necessary processing: e.g., generating resized thumbnails, compressing images, and perhaps scanning for inappropriate content if needed. The processing can be done using .NET image libraries (like ImageSharp) within a microservice or offloaded to cloud functions. A best practice is to avoid routing large file uploads through the main application server unnecessarily – we can generate pre-signed upload URLs for the client to upload images directly to storage [18], then inform our back-end of the new file location. This reduces load on our servers and speeds up uploads. All images will be associated with URLs stored in the database and delivered to clients via CDN, ensuring quick load times. By using a cloud storage + CDN approach, we ensure images are served from edge servers near the user, improving performance significantly [19]. Permissions and access control for these images will be handled via signed URLs or token-based access if needed (for example, event photos that are only visible to participants might use unguessable URLs or require an authorized request to fetch).

- **Authentication & Authorization:** The platform will implement a robust auth system to secure user data and functionalities. We will adopt **JWT (JSON Web Tokens)** for stateless authentication between the SPA and the back-end services. When a user logs in (via email/password or via Google/Facebook OAuth), the back-end Auth service will issue a signed JWT that the Angular client will store (likely in a secure HTTP-only cookie or local storage with proper precautions). This JWT will be sent with subsequent API requests to authenticate the user. Using JWT is a standard approach for SPAs and allows the back-end to verify the token without server-side session storage. We will follow security best practices for JWT handling: enforce short token lifetimes with refresh token rotation, use HTTPS only, and sign tokens using strong algorithms (preferably RSA asymmetric signing for extra security) [20]. For implementing OAuth2 social logins, we can integrate with Google's and Facebook's OAuth endpoints either directly or via an identity integration (for example, Firebase Auth or Auth0 can simplify this). On the back-end, **ASP.NET Core Identity** will be used for managing local accounts (email/password) and it can integrate with external logins. The new system will ensure that **authorization** (user roles/permissions) is in place as well – for instance, certain admin actions or event management features might require elevated rights. JWTs will carry role/claim information to enforce this on the server side. We will also secure our microservices (if there are multiple) in a **zero-trust** fashion – each service will validate tokens and not implicitly trust internal network calls. The use of industry-standard protocols and tokens will make the system secure and also interoperable if we later introduce a mobile app or other client. *Security is non-negotiable* – we will protect endpoints and enforce auth rigorously, using JWT + ASP.NET Core Identity or IdentityServer for a robust OAuth2/OpenID Connect solution [21]. All sensitive data (like passwords, API keys, JWT signing keys) will be stored securely (in environment configs or vaults, not in code) [22].

- **Best Practices & Architecture Considerations:** Throughout the migration, we will adhere to software engineering best practices to ensure the system is maintainable and robust. This includes: writing **automated tests** (unit tests for business logic, integration tests for service endpoints, and end-to-end tests for critical user flows) to catch regressions early; implementing comprehensive **logging and monitoring** (each service will output structured logs; we will aggregate logs and metrics using a tool like Elastic Stack or a cloud monitoring service, enabling us to trace issues across services); and maintaining infrastructure-as-code for repeatable deployments. We will enforce code style and perform code reviews for all changes. In design, we aim for *high cohesion and low coupling* – services are independent, communicate via well-defined APIs, and can be developed/deployed in isolation [6] [5]. We will avoid creating too many tiny services initially (to manage complexity) [23], instead starting with a reasonable set of core services and splitting further only when needed. Each microservice will encapsulate a **bounded context** of the domain (for example, the Event service deals with events creation/joining logic and its own data). For inter-service communication, wherever possible we'll use asynchronous messaging patterns to reduce tight coupling (for example, publishing an event "UserJoinedEvent" that other services can listen to, rather than direct HTTP calls for every interaction). We'll also include **resilience patterns** like timeouts and retries (using libraries like Polly in .NET) to handle network hiccups gracefully [24]. In the front-end, we will follow Angular's style guide and leverage features like **lazy loading** of modules and **ahead-of-time compilation** to keep the app fast and lightweight [25]. The front-end will be built as a **Progressive Web App (PWA)**, enabling features like offline support and push notifications down the line (this was hinted by adding service workers in earlier plans [26]). By using modern Angular, we also get built-in XSS protection and a robust framework for state management (we may introduce NgRx or a simpler state service for complex state). Lastly, the architecture will incorporate **DevOps best practices** from day one: everything from build to deployment is automated, and we practice continuous delivery of incremental improvements. Monitoring and alerting will be set up (using tools like Prometheus/Grafana or Application Insights) so we can proactively detect issues in production. By following these best practices – decoupling, rigorous testing, continuous monitoring – we aim to create a maintainable and resilient solution that can grow with our user base [27] [28].

In summary, the migration will transform Yapidoo's underpinnings while preserving and enhancing its core idea: connecting people to activities in real life. The **concepts outlined above** – CI/CD automation, microservices on Kubernetes, cloud service integration, multi-language support, robust image and auth handling, and sound architectural principles – provide an overview of how we will rebuild the platform for longevity and scalability. This foundation will set the stage for a successful implementation, which we detail in the roadmap and task breakdown that follow.

Roadmap with Key Milestones

The migration will be executed in phased milestones, each focusing on a set of goals and deliverables. Below is a high-level roadmap with key milestones, in logical order. Each milestone builds on the previous, ensuring that we achieve a working product at each stage before moving on. (Dates can be determined based on team capacity; here we focus on sequence and outcomes.)

Milestone 1: Foundation Setup – DevOps & Environment

Goal: Establish the foundational infrastructure for development and deployment, so the team can work efficiently on the new tech stack. In this phase, we set up the "plumbing" that everything else will rely on.

- **Source Control & Repositories:** Initialize new git repositories for the project (monorepo or separate repos for back-end and front-end). Migrate the existing code (if any) into the new repo structure as a reference, and prepare branching strategies. This also involves setting up code review processes (e.g. pull request templates, CI triggers on PR).
- **CI Pipeline Configuration:** Set up continuous integration pipelines that automatically build and test the code. For example, configure GitHub Actions or Jenkins pipelines to build the .NET services (restore NuGet, run unit tests, build artifacts) and the Angular app (npm install, run tests, build production bundle). Ensure that running the test suites and linters is part of the pipeline so we catch issues early. The pipeline should produce packaged artifacts (DLLs or Docker images) on successful builds [29] .
- **Dockerization of Services:** Write **Dockerfiles** for the back-end services and front-end app. Early on, even before all code is written, create basic Docker images (for example, an ASP.NET base image serving a "Hello World" API, and an Nginx or Node.js-based image to serve a sample Angular app) to verify our container setup. This ensures that the team can containerize new code from the start. We will use multi-stage builds for efficiency (build stage and runtime stage) [30] .
- **Kubernetes Cluster & Deployment Scripts:** Set up a Kubernetes cluster in a cloud environment. For this project, we consider using **Google Kubernetes Engine (GKE)** on Google Cloud for deployment. In this milestone, we'll provision a dev/test Kubernetes cluster. We also create Kubernetes manifest files or Helm charts for deploying our services. Initially, this can be very simple (one deployment and one service for the API, one deployment and service for the web app). The idea is to get the basic continuous deployment working: e.g., after CI builds an image, a CD step applies the K8s manifests to deploy that image to the cluster. By the end of this milestone, we aim to have a "Hello World" version of the new architecture running on K8s – for instance, an Angular app served from one container calling a trivial endpoint of the .NET API in another container, all deployed via our pipeline. This proves that our pipeline, Docker, and K8s setup works end-to-end. *(Success Metric:* Developers can merge code and see an updated container running in the test environment automatically).

Milestone 2: Core Architecture & Framework Implementation

Goal: Build out the core architecture of the application without yet adding all the business features. This includes establishing the structure of microservices, implementing fundamental capabilities like identity, and scaffolding the front-end with routing and state management. Essentially, we create the skeleton of the app (both back-end and front-end) such that future features can plug into it.

- **Back-end Core Services Creation:** Implement the initial set of microservices using .NET (likely ASP.NET Core 7 or 8). At minimum, start with the **Authentication/Identity Service** and one or two other core services (for example, an **Event Service** to handle event creation logic). Define the data models and set up databases for each service (for now, perhaps using a simple SQL database instance with separate schemas, or multiple database instances). Implement fundamental endpoints such as registration/login (`/api/v1/account/register` etc.) and health-check endpoints for services. We will also integrate **ASP.NET Identity** or similar for managing user accounts in the Auth service. This milestone includes setting up cross-cutting concerns in the services: centralized logging, error handling middleware, JWT authentication setup, and CORS configuration so the Angular app can call the APIs. The Auth service will be configured to issue JWTs upon successful login. By the end of this milestone, the back-end should have the ability to register a new user and return a JWT, and perhaps a protected test endpoint that requires the JWT (proving that auth works).
- **Front-end Application Scaffolding:** Create a new Angular application (using latest Angular CLI). Set up the project structure (modules, components) following best practices (e.g. a core module, shared module, feature modules for different sections of the app). Implement the routing structure for the major pages (e.g. routes for Home/Landing, Login, Register, Event Feed, Profile, etc.) even if the pages themselves are mostly placeholders initially. Include a basic global state management solution (could be NgRx store or simpler BehaviorSubjects for user session state) to handle things like the current logged-in user info. Also, integrate Angular Material or a UI library if we plan to use one for quicker UI development. By milestone's end, the front-end should be able to navigate between a few pages and perform a test login flow: e.g., a user can go to the login page, input credentials, the app calls the Auth service API, receives a token, and stores it (in memory or localStorage), and the state reflects that the user is logged in. It might then show a placeholder "Dashboard" page accessible only if logged in. This proves that the front-end and back-end can communicate securely.
- **Internationalization Setup:** Before the front-end grows too large, integrate the i18n library. For Angular, we decide whether to use the built-in i18n (`@angular/localize`) or a runtime library like ngx-translate. At this stage, set up the infrastructure for translations: e.g., have a translations file for English and one for another language (Ukrainian) with a few sample translated strings. Implement a language toggle in the UI and verify that switching languages changes the text. Also ensure the app can detect the browser language on load and set default accordingly [8] . This milestone doesn't require every string to be translated yet, but the mechanism should be in place.
- **Static Pages and Basic UI:** Implement the basic landing page and maybe an about page or contact page as static content, just to have some content in the app. This is also a chance to integrate the styling framework (CSS/Sass setup, maybe import designs from Figma or use a theme). The landing page can include the login/register forms created earlier. At this point, we may not have real event data to display, so the focus is on layout and navigation. The landing page will eventually contain links or sections (as described in the concept documents, like sections for About, Team, etc.), so structuring it now is good.
  (Success Metric: By the end of Milestone 2, we have a running Angular app that can authenticate with the new backend. The app skeleton and a couple of real pages are in place, though full functionality isn't there yet. The basic multi-language toggle works. The system is deployed on the test K8s cluster, meaning one can actually visit the test URL, see the landing page, register a user, and log in (with the interactions going through the new microservices and database).)

Milestone 3: Implement Essential Features (MVP Parity)

Goal: Develop the core features of Yapidoo so that the new system reaches MVP feature parity with the old system (Yalldoo). This milestone focuses on the main use-cases: event creation, browsing/joining events, and social interactions (friends), as well as completing the user profile management. Essentially, we build the primary product features on top of the established framework.

- **Event Management Features:** Create the **Event Service** (if not started earlier) which handles creating events, editing events, viewing event details, and joining events. Define the API endpoints such as `POST /api/v1/events` (to create an event), `GET /api/v1/events?filters` (to search or list events), `POST /api/v1/events/{id}/join` (to join an event), etc., according to the product requirements. Implement business rules like event visibility (private events vs public), capacity limits, and any "confirmation" logic (some events might require host approval to join, per initial idea). On the data side, create the database tables for events, and possibly a join-table for participants of events. Implement logic so that when a user joins an event, they are associated with it. Also include the "event feed" concept: e.g., an endpoint to get a user's personalized feed (combining events by their friends and public events nearby). This might be part of the Event service or a separate Feed service, but initially it can be a query on events filtered by friend relationships and location.
- **Front-end for Events:** Develop Angular components for event creation and event listing. For creation: build a multi-step form (the documents suggest a three-step form for creating an event [31], collecting info like name, time, location, details, privacy settings, etc.). Include features like date/time pickers for event time, location selection (possibly integrate Google Places auto-complete into a location field), and uploading a cover image for the event. Validate inputs (no empty required fields, etc.). For browsing events: create a page that shows a list of events. This could be split into "All Events", "Friends' Events", and "My Events" tabs as described [32] [33]. Implement sorting or filtering controls (by time, by newest, by location) similar to what was envisioned. When a user clicks an event, navigate to an Event Details page showing full info and an option to join or leave the event. If the user is the creator, they should see editing options. This front-end work will consume the Event API implemented above.
- **Friend & Social Features:** Implement the **Friend (Social) Service** to handle friend requests, friend connections, and any social feed logic. This involves endpoints like sending a friend request, accepting/declining requests, listing friends, etc. The data model will include a friendship relationship table. On the front-end, add UI components for social interactions: e.g., the ability to search or invite friends, a page or sidebar that lists pending friend requests (with accept/decline buttons), and a friend list page. Also, incorporate friend-based filtering on events (for the "Friends' Events" feed mentioned above, which shows events created by your friends [34]). Possibly integrate this with the profile page – e.g., viewing someone's profile could allow sending a friend request. By the end of this, users in the new system can add each other as friends and see friends' events.
- **User Profile & Preferences:** Complete the **Profile** section for users. Since basic registration was done earlier, now implement the ability for a user to view and update their profile info (name, photo, bio, location, interests, etc.). This might be part of the Auth service or a separate Profile service. Build an API like `GET/PUT /api/v1/users/me` to get or update profile data. On the front-end, create a Profile page where users can edit their info and upload a profile picture. This is also where language preference could be set if we allow user to save preferred language (though language might just follow the UI choice). Additionally, consider implementing the "preferences" or interests selection that was described in the concept (users selecting their hobbies/interests, which later can feed into recommendations) [35]. This could be as simple as a multi-select of interest tags stored in the user profile.

- **Notification/Email Feature (if part of MVP):** Implement basic notifications such as email verification and event invitations. For example, ensure that when a user registers via email/password, a verification email is sent (the earlier tasks list had an email confirmation flow [36] [37]). We can integrate a service like SendGrid or use Firebase Cloud Functions to send emails. Set up the back-end to handle email confirmation links (an endpoint that verifies a token and activates the account). Also, if time permits in this milestone, implement real-time notifications for events (e.g., when someone joins your event, you get notified). This could be done via simple means like sending an email or by preparing for push notifications (perhaps leveraging Firebase Cloud Messaging in the future).
- **Testing of Features:** As features are built, write tests (unit tests for logic, and integration tests for APIs). Also, start usability testing within the team for flows like event creation and joining, to ensure the UX matches expectations. We will deploy these features to a staging environment (maybe the same test K8s cluster) so that a small group (including the team and some pilot users) can try out the new system and provide feedback.
  (Success Metric: Milestone 3 is essentially reaching a **Minimum Viable Product** state. By its end, a user should be able to perform all critical actions: sign up, log in, fill out profile, add friends, create an event, join someone else's event, and see events on a feed. The new application would be capable of the basic value proposition of Yapidoo. We should be able to demonstrate a full user journey in the new system. At this point, the new system could potentially be used by end-users in a beta. Key success indicators include: no critical missing pieces in the core flow, and the system remains stable with a small number of test users performing actions.)

Milestone 4: Enhancements – Integration & Optimization

Goal: Add the supporting and advanced features that will make the platform robust and polished. This includes integrating third-party services (Google APIs, Firebase), implementing the remaining nice-to-have features, and optimizing performance and security before launch.

- **Google/Firebase Integration:** Integrate the platform with Google services as planned. For **Google Login**, configure the front-end to use Google's OAuth 2.0 (possibly via the Google Identity SDK) so users can sign in with their Google account. This will involve setting up OAuth credentials in Google API Console and then, on success, exchanging the Google token for our JWT on the back-end. For **Google Maps/Places**: incorporate a map or place picker in the event creation form (e.g., allow the user to drop a pin or search for a place when specifying event location). Use the Google Maps JavaScript API on the front-end for this and store coordinates or place IDs in the event data. Additionally, if using **Firebase** for any functionality: for instance, if we want to implement chat for event participants, we might use Firebase Realtime Database or Firestore for messaging since it can simplify real-time updates. Alternatively, we could stand up our own signalR or WebSocket service for chat, but using Firebase could expedite it. Decide on whether real-time chat is in scope now; if yes, implement a basic chat where participants of an event can send messages (accessible on the event detail page) – this could use Firebase under the hood and the Angular app would just use Firebase SDK for real-time updates. Another Firebase feature to consider now is **Firebase Cloud Messaging (FCM)** for push notifications: set up the front-end to register for push notifications (especially if we make it a PWA), and use FCM to push notifications like "Someone joined your event" or "New friend request" to users. The back-end can call the FCM API (using Firebase Admin SDK in .NET [38]) to send these notifications. All of these integrations will be configured and tested in this phase.
- **Multi-Language Content Completion:** Go through the UI and ensure that **all** user-facing text has been externalized for translation. Work with translators (or use translation services) to produce the text in the supported languages (e.g., Ukrainian, English, etc.). Populate the translation files. Test the app thoroughly in each language to catch any layout issues (for

example, longer text in one language might overflow a button). Ensure that emails or notifications sent to users are also localized as appropriate (possibly defaulting to a language based on user preference). This milestone delivers full multi-language readiness for launch.

- **Image Handling & Optimization:** Finalize the image upload and processing pipeline. By now, users can upload images for profiles and events; we need to ensure these are stored and retrieved efficiently. Implement any remaining processing – e.g., create thumbnail versions of event photos for faster loading in lists. We might write a background worker (or use a cloud function) to generate these after an upload. Also, integrate a CDN (if not already) for serving the images. Test image upload and delivery performance. If any limits or issues are found (like very large images causing slow loads), consider adding client-side resizing (the Angular app can resize images before uploading using canvas, for example) in addition to server-side processing. Also, enforce storage security: ensure that private images (if any) are not publicly accessible without authorization.
- **Performance Tuning & Security Audits:** With all major features in place, conduct a round of performance optimization. This includes load testing the API (simulate many concurrent users creating or joining events) to identify bottlenecks. Scale up the K8s deployment as needed (increase replicas, maybe test auto-scaling settings). Optimize database queries (add indexes or caching for frequently accessed data like event feeds). On the front-end, use Angular's build analyzer to ensure bundle sizes are trimmed (remove any unused dependencies). Enable production optimizations like Angular's Ahead-of-Time compilation and tree-shaking (if not already). Also, set up caching headers for static content and API responses where applicable (e.g., CDN caching for images, maybe HTTP caching for certain GET endpoints).
- **Security review:** Double-check that all sensitive API routes are protected (e.g., you cannot join an event without being authenticated, etc.). Penetration-test the app for common vulnerabilities (XSS, SQL injection – though using parameterized queries/ORM should mitigate SQLi, CSRF – our use of JWT in header avoids CSRF for the API, but if we use cookies we need CSRF tokens). Ensure that the JWT implementation uses secure cookies if applicable, and refresh token flows are correct (possibly implement refresh token endpoint if not already). Also review cloud security: e.g., ensure the storage bucket is not world-readable, etc. It might be worthwhile to use a tool or service for a security audit.
  (Success Metric: By the end of Milestone 4, the platform is feature-complete and polished. All integrations are working (e.g., a user can log in with Google, select an event location on a map, get push notifications, etc.), the app is fully translated, and performance is tuned for the expected initial user load. We should feel confident that the product is ready for real users. Any beta testing feedback from milestone 3 is addressed in this phase as well, so the user experience is smooth.)

Milestone 5: Testing, UAT & Launch Preparation

Goal: Before going live, ensure the system is thoroughly tested end-to-end, get final user acceptance, and prepare for a smooth production launch.

- **Comprehensive Testing:** Conduct end-to-end testing scenarios covering all use cases: user registration (with email verification), social login, creating events, joining events, sending friend requests, etc. Use a staging environment that mirrors production (the staging could be our test K8s cluster with production-like configuration). Have team members or a small group of external beta users perform UAT (User Acceptance Testing). Collect any bug reports or UX issues and fix them now. Also test edge cases – e.g., what happens if an event is full and another user tries to join, or if two users try to join at the same time (race conditions), what if the database or an external API is temporarily down (does the app handle it gracefully?). Verify the system's behavior under load (perhaps simulate a peak load such as 1000 concurrent users to see if auto-scaling kicks in and system remains stable).

- **Data Migration and Seed:** If there is existing data from a legacy system (say Yalldoo had some users or events in an older database), plan the migration of that data. Write migration scripts or programs to transfer users (with passwords or ask them to reset passwords if we cannot migrate those securely), migrate friend relationships, and maybe upcoming events into the new system. This may happen just before launch. If no legacy data, we might seed the database with some initial data (e.g., sample categories of events, maybe a few example events in major cities to not have an empty feed on day one). Ensure that after migration, all foreign keys and relationships are consistent in the new databases.
- **Monitoring & Logging Setup:** Configure production monitoring: set up dashboards (for CPU, memory, error rates, etc. in the K8s cluster). Configure alerts for critical conditions (e.g., if an API starts failing or if response time exceeds a threshold). Set up an error tracking service (like Sentry or Application Insights) to catch any front-end or back-end exceptions in production. Verify that logs from all microservices are aggregating in whatever central log system we chose, so we can quickly diagnose issues post-launch.
- **Final Security Checks:** Ensure SSL certificates are in place for the production domain (we will likely use HTTPS by default in Kubernetes ingress or a cloud load balancer). Double-check that no test accounts or backdoor admin endpoints are present. If using third-party scripts (analytics, etc.), make sure they are updated and secure. Possibly run an external vulnerability scan.
- **Deployment Plan:** Formulate the plan for going live. This might involve provisioning a production-grade K8s cluster (if we haven't already – possibly we upgrade our staging cluster to production or create a new one for prod). Decide on a go-live date and a strategy (blue-green deployment, or simply deploying v1.0 to prod and switching DNS). Ensure rollback plans are in place: e.g., if something goes wrong after launch, have a way to either fix forward quickly or temporarily take the system down gracefully (with a maintenance page). Because this is a brand new system, we won't have an "old version" to fall back to in production; however, if the old system was live and had users, coordinate the cut-over carefully (maybe freeze the old system, migrate data, then switch on the new system). Inform any existing users (via email) of expected downtime or changes, if applicable.
  (Success Metric: Milestone 5 concludes with a green light for launch – all tests are passing, stakeholders have approved the system, and we have confidence in the system's stability and security. At this point, the only step remaining is to actually open it up to users.)

Milestone 6: Production Launch & Post-Launch

Goal: Launch the new Yapidoo platform to production and closely monitor its performance, followed by iterative improvements.

- **Production Launch:** Deploy the code to the production cluster, ensuring all configurations are set for production (database connection strings, API keys, domain names, etc.). Do this during a low-traffic period if migrating from an old system. Once deployed, perform smoke tests in production to confirm everything is running (create a test event, etc., then remove it). Announce the launch to the user community. If this is the first launch of the platform (no existing users), simply start marketing at this point to drive users to the site. If replacing an old system, decommission the old environment after verifying the new one is stable.
- **Post-Launch Monitoring:** In the first days/weeks after launch, keep a close eye on system metrics and user feedback. Watch for any spikes in errors or performance lags and address them immediately (e.g., if certain queries are slow with real user data, optimize indexes or add caching). Scale the system as needed if user growth is faster than expected – Kubernetes makes it straightforward to increase replicas or node counts. Also monitor costs on the cloud; optimize

resource usage if any component is consuming too much (e.g., maybe tweak auto-scaling thresholds or use more cost-effective instance types).

- **Collect Feedback & Continuous Improvement:** Gather feedback from real users. Perhaps integrate an analytics tool (if not already) like Google Analytics or Application Insights to see user behavior (which features are used, where drop-offs happen). Use this data to plan further improvements or new features. Users might request features like event reminders, better search filters, mobile app, etc. We will enter a continuous development cycle to prioritize these. Trello boards or Jira can be updated with new user stories and the development team can iterate in sprints. The architecture we put in place will allow us to roll out updates regularly via the CI/CD pipeline (practicing continuous deployment). We will also address any technical debt left during the rushed development (if any hacks or shortcuts were made, now is time to refactor given actual usage patterns).
- **Marketing & Scaling (ongoing):** Although not a technical step, part of post-launch is ensuring the platform scales socially – integration with social media for sharing events, perhaps leveraging SEO for public event pages. We should verify that when users share event links, the site has proper meta tags (OG tags) so that it unfurls nicely on Facebook/Twitter. If any scaling issues arise from usage (like a surge of new users causing something to break), the tech team will use the architecture's flexibility (microservices, horizontal scaling) to alleviate them. This milestone is essentially an ongoing phase after initial release.
  (Success Metric: The platform is live and stable. We measure success in real user terms now: number of events created, user engagement, growth rate. From a tech perspective, success means the system handles the traffic without major issues, and any minor issues are detected and resolved quickly via monitoring and our agile response process. The team is able to continuously improve the product after launch without significant downtime, thanks to the solid DevOps foundation.)

This roadmap provides a structured path from preparatory work through to launch. Each milestone is a checkpoint with specific outcomes that collectively ensure a successful migration and modernization of Yapidoo. By following this roadmap, we mitigate risks (through early setup of pipeline and gradual feature development), validate at each step (with testing and partial deployments), and keep the project aligned with its ultimate goals. The next section will break down specific tasks in detail and indicate their dependencies, to facilitate tracking in project management tools like Trello or Jira.

Tasks with Interdependent Linkage (for Trello Board)

Below is a detailed task breakdown for the migration project, including dependencies and parallelization notes. Each task corresponds to a unit of work that could be tracked on a Trello card (or Jira ticket). Where applicable, we indicate if a task is blocked by another (i.e., should start only after a dependency) or if it can proceed in parallel with others. This will help in sequencing the work and identifying critical path vs. independent tasks.

1. **Set up Git Repositories and Codebase Structure** – *Dependencies:* None. *(This task is foundational and can start immediately.)* Create a new repository (or repositories) for the project. Organize it to contain the back-end services (e.g., separate folders/projects for Auth, Event, etc.) and the front-end app. Initialize solution files, and set up basic readme/build instructions. This provides the scaffolding for all development to come.

2. **Configure CI Pipeline (Build & Test)** – *Dependencies:* None. *(Can be done in parallel with task #1, as they complement each other.)* Choose a CI tool (GitHub Actions, GitLab CI, Jenkins, etc.) and write pipeline scripts to automatically build and run tests for the .NET services and the Angular app. For example, set up jobs to restore packages, compile the code, run unit tests, and produce build artifacts. This task ensures every commit is validated. It is independent of writing application code and thus can be done alongside initial codebase setup. (No blockers, runs in parallel with initial dev work.)

3. **Provision Kubernetes Cluster (Dev/Test)** – *Dependencies:* None. *(Can be done in parallel with tasks #1 and #2.)* Create a Kubernetes cluster in the chosen environment (e.g., GKE on Google Cloud). This involves setting up the cluster itself and networking (and possibly configuring a container registry for images). Since this is an infrastructure task, it doesn't depend on application code. It can proceed in parallel, so that when the first images are ready, we have a place to deploy them.

4. **Dockerize Base Application Components** – *Dependencies:* #1 (code structure in place). *(Partially parallel with #2 and #3: one can start writing Dockerfiles once we have at least a sample project in the repo.)* Write Dockerfiles for a sample .NET service and the Angular app. This includes choosing appropriate base images (e.g., `mcr.microsoft.com/dotnet/aspnet:7.0` for runtime, `:7.0-sdk` for build stage, and maybe `node:18-alpine` for Angular build stage). This task produces Docker images and should hook into the CI pipeline. It's listed after #1 because we need some project to build, but it can be done concurrently with pipeline config (#2) since defining the Dockerfile is independent of the CI tool (though we will integrate it into CI once ready).

5. **Set up CD Pipeline (Deploy to K8s)** – *Dependencies:* #2 (CI pipeline), #3 (cluster), and #4 (Docker images).* After CI is building images and the cluster exists, configure Continuous Deployment. This involves writing deployment scripts or Helm charts, and then automating the deployment via the pipeline. For instance, after a successful build, have the pipeline push the Docker images to a registry and run `kubectl apply` (or Helm) to deploy to the cluster. This task is blocked until we have something to deploy (Docker images) and somewhere to deploy (cluster), hence it comes after those. Once set up, any new image build will be rolled out to the dev environment. [12]

6. **Design Database Schema(s) & Set Up Databases** – *Dependencies:* None. *(This can start early, even before coding, and in parallel with infrastructure tasks.)* Plan the data model for the new system: determine tables for Users, Events, FriendConnections, etc, respecting a per-service schema. Set up a development database instance (or multiple if using separate DB per service; could be MySQL, PostgreSQL or SQL Server depending on preference). This task can proceed independently and earlier so that developers know the target schema. It might produce an initial SQL migration or Entity Framework migrations for each service. No blockers, and doing this early helps subsequent implementation tasks.

7. **Implement Authentication Service (Backend)** – *Dependencies:* #1 (project setup) and #6 (knowledge of user schema).* Develop the Auth service using .NET. This includes models for User, roles, etc., and endpoints for register, login, logout, refresh token. Integrate JWT token issuance. Use ASP.NET Identity or custom implementation as decided. This task is dependent on having the project structure in place to add a new service (done in #1) and ideally knowing the database structure for users (#6). Once those are in place, this coding task can proceed. It may also overlap with front-end auth implementation (task #9) – they can be developed in tandem, using agreed API contracts.

8. **Implement Core Event Service (Backend)** – *Dependencies:* #1 and #6.* Develop the Event Management microservice in .NET. Implement endpoints like CreateEvent, GetEvent, ListEvents (with filtering), JoinEvent, etc. This task relies on the general setup (#1) and the database schema for events (#6). It is not strictly blocked by the Auth service (#7); a developer can build the Event service in parallel to Auth, using stubbed auth (like a dummy user ID) until Auth is ready. However, full testing will eventually require Auth for security. We consider it parallelizable with #7, as different team members can work on different services concurrently, as long as they define how they will integrate (e.g., Event service will trust a JWT's user ID claim for authentication).

9. **Develop Front-end Authentication & Landing Pages** – *Dependencies:* #1 (Angular project scaffold), and parallel with #7.* On the Angular app, create the pages for Login, Registration, and the basic landing page. Hook up the forms to call the Auth service API (task #7). This includes setting up an HTTP service for API calls, managing JWT storage (e.g., in memory or localStorage), and updating UI on login state. This task can start once the Angular project exists (#1 gives structure) and does not need Auth service fully done if we use mock responses initially. It can run in parallel with #7, coordinating on the API format. Once #7 (Auth API) is ready, integrate it for real. This task is essentially front-end development to match the Auth functionality and ensure users can log in from the UI.

10. **Implement Profile & User Management Features (Backend)** – *Dependencies:* #7 (Auth basic functionality).* Extend the Auth service or create a Profile service to support user profiles. Implement endpoints for getting user profile, updating profile, uploading profile picture, etc. Also, implement email verification and password reset flows if required (sending emails via a background task or directly for now). This task logically comes after basic auth (#7) because it builds on the existence of user accounts. It could be parallel to #8 or #9, but we list it after #7 to ensure Auth is mostly done. It may also depend on having an email service configured (could tie into #16 later if using Firebase or another email solution). For now, block it on #7 completion to avoid confusion.

11. **Implement Front-end: Event Creation & Feed Pages** – *Dependencies:* #8 (Event service API design), and #9 (basic front-end structure).* Develop the Angular components for creating an event (the multi-step form) and for viewing lists of events (feed). This task should start after we have a sense of the Event API (#8) – at least the contract. It can be done in parallel with #8 if the team agrees on the API spec in advance (possibly using a stub JSON or OpenAPI). It also requires that the Angular app's basic layout and routing (#9) is in place. Once those are set, one can build the event-related pages. This includes integrating map auto-complete (which might depend on Google API – could tie into #16, but a placeholder can be used initially). Also, implement the event detail page and the join button which will call the join API. So, #11 is primarily front-end tasks relying on #8's outcome. Ideally, do #8 and #11 concurrently with close communication.

12. **Implement Social/Friend Service (Backend)** – *Dependencies:* #7 (users exist) and #6 (friend schema).* Develop the Friend or Social service that handles friendships. Endpoints might include: send friend request, respond to request, get friends list, get pending requests. This depends on user identities (#7) being in place and a planned schema for friend relationships (#6). It can be done in parallel with #8 and #10 in terms of coding, since it's a separate domain. However, it slightly depends on #7 because without a user system, friend relationships make no sense. But by the time #7 is done, #12 can proceed. It is not strictly blocking front-end event development (#11), so it can be parallel to that.

13. **Implement Front-end: Friend Management UI** – *Dependencies:* #12 (Friend service API), and somewhat on #9 (base UI).* Build UI elements for the social features. For example, a page or modal that shows incoming friend requests with accept/decline, a list of friends, and the ability to search users to add (if applicable). Also integrate friend data into other parts: e.g., mark events that were created by friends, or possibly filter feed by friends. This front-end work should start once the Friend service API (#12) is ready or at least defined. It can overlap with event front-end (#11), as different devs can handle different sections of the app. Ensure to test end-to-end: e.g., user A sends request, user B sees it and accepts, then both see each other in friends list.

14. **Set Up Image Storage and Upload API** – *Dependencies:* #8 (event service, if event images) and #10 (profile, if profile pics).* Now that core features are built, implement the image storage solution. This likely involves a few steps: create a Google Cloud Storage bucket (or Firebase Storage) for images, generate a service account/credentials for the back-end to use. In the back-end (perhaps in a shared utility service or within each service like Profile and Event), implement an endpoint to receive image uploads or to generate upload URLs. For example, an API `/api/v1/events/{id}/uploadImage` that returns a signed URL which the front-end can PUT an image to. Alternatively, allow direct upload through the API by accepting multipart form data. Given best practices, we might do the signed URL approach [18]. This task depends on event and profile being in place (#8 and #10) because those define where images attach. It can start as those near completion. Also, it ties into environment configuration (credentials for GCP), but no other tasks block it. It should be completed before we finalize the front-end image upload functionality (#15).

15. **Integrate Image Upload on Front-end** – *Dependencies:* #14 (image backend ready), and #11/#13 (UI where images are used).* Add functionality in the Angular app to handle image uploads for profile pictures and event images. This includes image selection (file input), preview, and calling the appropriate API (from #14) to upload. After upload, ensure the new image URL or ID is saved and the UI displays the uploaded image (perhaps with a refresh from the server or an immediate preview). This task is blocked by #14 because we need the back-end support. Once #14 is done, this is straightforward to implement in parallel to any remaining UI polishing.

16. **Google Services Integration (Maps, OAuth, Notifications)** – *Dependencies:* core features implemented (should be after #8, #11 for context) and possibly #14 (for Firebase config if using FCM).* This encompasses multiple integration points with Google:

    - **Google OAuth Login:** Set up the Google Developer Console project, obtain OAuth Client ID. In Angular, integrate Google Sign-In SDK so users can click “Login with Google”. On success, retrieve the Google token and send to our Auth service (we might need to implement an endpoint in Auth to handle Google login by verifying Google token and creating a JWT for our app). This likely depends on Auth (#7) but can be done after core Auth flows.
    - **Google Maps/Places:** Enable Google Maps JavaScript API, get API key. In the event creation form (front-end, #11), replace any placeholder location input with the Google Places Autocomplete field. Also possibly show a small map for selecting location. This integration should happen after the basic form works.
    - **Firebase setup (optional for notifications or chat):** If using Firebase Cloud Messaging for push notifications, configure Firebase project and add the FCM SDK in Angular. Get user permission for notifications, and set up the back-end to send messages (which might require storing FCM tokens for users). Similarly, if using any Firebase DB for chat, integrate that in event detail page to allow messaging.

    - **Analytics:** Optionally integrate Google Analytics or Firebase Analytics for usage tracking. This task can be started once the respective areas of the app are ready to accept integration. Google login can be done once Auth is done, maps once the event form is present, etc. There might not be strict blockers beyond those features existing. It's grouped as one because it's all Google-related, but in execution, it could be multiple tasks (one for OAuth, one for Maps, etc.). Here it's combined for overview. Ensure to test each integration thoroughly (e.g., logging in with Google returns a valid account in our system, map autocomplete returns coordinates and they're saved with the event, push notification test from server reaches the browser, etc.).

17. **Write Unit and Integration Tests (ongoing)** – *Dependencies:* ongoing, but fully cover after features built.* While testing is an ongoing activity with each feature, this task is to ensure by the end we have good test coverage. It may involve writing additional unit tests for complex business logic (e.g., cannot join an event if it's full or past its date, etc.) and integration tests that simulate API calls (possibly using an in-memory test server for .NET or postman test suites). This task isn't blocked by any single item; rather it should start once some features are ready and continue throughout. We list it here to ensure time is allocated to beef up testing after implementing features. By the end of the project, we want a robust automated test suite as part of CI [29] . No strict dependency, but naturally it happens after or alongside feature development.

18. **Performance Testing & Tuning** – *Dependencies:* core features complete (after #15, #16).* Once most functionality is in place, perform load testing and optimize. This involves using tools (like JMeter or k6) to simulate many users and see how the system behaves. Identify slow endpoints or heavy queries. Optimize code, add caching, or scale resources as needed. Also test front-end performance (Lighthouse audits, etc.). This task depends on the application being mostly built to test it thoroughly. It's not blocked by Google integration (#16) necessarily, but it should happen toward the end of development. It can run in parallel with final integration and bug-fixing tasks.

19. **UAT and Bug Fixing** – *Dependencies:* all major features done (post #16).* Coordinate a User Acceptance Testing phase where either internal team members or a closed beta group use the system as end-users. Collect feedback and log issues. This task entails fixing any discovered bugs or UX problems. It is dependent on having a working end-to-end system (so essentially after implementing everything above). But it can overlap with #18 (performance tuning) and some of #16 (if UAT starts while final integrations are being wrapped up). Treat UAT as the final polish stage where all blockers for launch are resolved.

20. **Deployment & Launch Prep** – *Dependencies:* #5 (CD in place) and all features tested.* Prepare production release. This includes setting up production configuration (in K8s, perhaps a separate namespace or cluster with production settings like proper domain, enabling SSL on ingress, scaling parameters). Also, do a final security audit at this stage. This task is at the very end, dependent on everything else being done and verified. It's the culmination where we might do a dry-run deployment to ensure no misconfigurations. Once this is done, we can officially launch. (In a Trello context, this might be a card for "Go Live!" with a checklist of pre-launch steps.)

Each of these tasks would have additional details and sub-tasks in practice, but as listed they cover the major pieces of work. The dependencies are summarized as:

- Tasks **1, 2, 3** are initial setup tasks with no prerequisites (can run concurrently).
- Task **4** (Dockerization) follows task 1 (needs code structure) but can overlap with 2 and 3.
- Task **5** (CD pipeline) waits for 2, 3, 4 to be in place (needs code build + cluster + images).
- Task **6** (DB design) is independent and can start early.

- Core development tasks **7, 8, 9, 10, 12** (Auth, Event, Front-end basic, Profile, Friend backend) all depend on initial setup (task 1, and schema from task 6). These can be done somewhat in parallel by different team members, with Auth (#7) being a prerequisite for things like Profile (#10) and influencing others. Friend (#12) ideally after Auth since it uses user data.
- Front-end feature tasks **11** (Event UI) depends on #8 (Event API) and #9 (base UI ready); **13** (Friend UI) depends on #12 (Friend API) and base UI. They can parallel each other and to some extent with #9 once APIs are ready.
- Task **14** (Image backend) depends on having at least event or profile logic to attach to (so after those backend tasks exist).
- Task **15** (Image front-end) depends on #14 done.
- Task **16** (Google integration) depends on relevant core features (Auth for OAuth, Event for Maps, etc.) and comes after those are functional.
- Tasks **17, 18, 19** (Testing, Perf, UAT) are ongoing/finalizing tasks after implementation (they don't produce new features, but ensure quality). They happen near the end, with no new dependencies except the system being feature-complete.
- Task **20** (Launch prep) is last, after everything including testing is done.

By visualizing these dependencies, we can identify critical path items: for instance, setting up CI/CD early (tasks 2, 5) is critical to enable fast development feedback; Auth service (#7) is critical for many user-related features; and finishing all integration (task 16) and testing (17-19) is critical before launch. Many tasks (like developing different services and front-end pages) can proceed in parallel, which will shorten the overall project timeline if done by a team.

This task breakdown with dependencies will guide the project management. In Trello, we could label tasks by category (DevOps, Backend, Frontend, Testing, etc.) and use attachments or checklists to denote blockers. For example, a Trello card "Implement Event Service" could have a checklist item "Auth service API ready" if that's a prerequisite. Likewise, cards can be moved to "Blocked" column if waiting on another task. Using the dependency info above, the team can sequence work and also tackle independent tasks concurrently to accelerate progress.

Each task card can also include relevant details (e.g., acceptance criteria or links to design docs from earlier in the folder) to ensure clarity. By maintaining these interdependencies explicitly, we ensure that, for example, front-end work isn't done on a feature before its API exists (unless mocking), or that we don't attempt to deploy before the cluster is configured, and so on. This prevents wasted effort and aligns development with the roadmap milestones.

Lastly, as tasks are completed, their status on the board (to-do, doing, done) will give a clear picture of progress. Team members should communicate when a blocker is resolved (e.g., "Auth API is done, now Event UI can integrate with real data"). This synchronized approach will help the migration project proceed smoothly and finish on time, delivering a fully modernized Yapidoo service.

---

[1] [2] [3] Yalldoo letter
https://docs.google.com/document/d/1YBu6KLb7JOC5-8LUhfdflbUfiF_rzpxxYjnYwcAHBs0

[4] Migrating Angular & .NET Docker Environment to Kubernetes ☸ | by Anvesh Muppeda | Medium
https://medium.com/@muppedaanvesh/migrating-angular-net-docker-environment-to-kubernetes-8f010b597b91

[5] [6] [7] [10] [13] [21] [22] [23] [24] [29] [30] Best Practices for Microservices in .NET | by Xperture Solutions | Mar, 2025 | Medium
https://medium.com/@xperturesolutions/best-practices-for-microservices-in-net-cc3005803005

[8] [9] [31] [32] [33] [34] [35] Yalldoo описание
https://docs.google.com/document/d/1JM0aR-pAxO_H0wb2b50BdyfE-Z2tXtcGPWRFxxfXHQE

[11] [12] From 0 to Kubernetes using .NET Core, Angular e Jenkins - Codemotion Magazine
https://www.codemotion.com/magazine/frontend/web-developer/from-0-to-kubernetes-using-net-core-angular-e-jenkins/

[14] [26] [36] [37] Yalldoo - Workflow Diagrams
https://docs.google.com/presentation/d/1qqoN0rlHMgGW0qUGrDNYugfJM4PdDWIggsktplQbA70

[15] Internationalization • Overview • Angular
https://angular.dev/guide/i18n

[16] A Complete Guide To Angular Multilingual Application (i18n)
https://medium.com/angular-in-depth/a-complete-guide-to-angular-multilingual-application-91f431f0f12c

[17] [19] Best way to store product images - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/1319558/best-way-to-store-product-images

[18] A Developer's Guide to Uploading Images: Best Practices and ...
https://medium.com/@digveshparab123/a-developers-guide-to-uploading-images-best-practices-and-methods-2ecb20133928

[20] JWT Authentication in .NET 8: A Complete Guide for Secure and ...
https://medium.com/@solomongetachew112/jwt-authentication-in-net-8-a-complete-guide-for-secure-and-scalable-applications-6281e5e8667c

[25] [28] Elevating Your High-Traffic App: Integrating .NET Aspire with Docker, Kubernetes, and Angular | by Murat Aslan | DevOps.dev
https://blog.devops.dev/elevating-your-high-traffic-app-integrating-net-aspire-with-docker-kubernetes-and-angular-a9de13dc8cb3?gi=c7c73814f2bf

[27] Modern Microservices with .NET 8: Architectures, Tools, and Best ...
https://medium.com/@anderson.buenogod/modern-microservices-with-net-8-architectures-tools-and-best-practices-405ba73561b2

[38] Firebase Admin .NET SDK - GitHub
https://github.com/firebase/firebase-admin-dotnet