Risk Management Plan for Sprint 3 and Sprint 4

Context and Objectives

The team is developing a web-based application for managing pre-assessment quizzes and helper assignments at a university cyber-fair. Sprint 3 focused on establishing core technical foundations (database schema, development environment and CI/CD) while addressing foreseen risks from earlier sprints. Sprint 4 built upon this by delivering helper-management functionality and tackling the complexity of the existing ASP.NET code base. The risk management plan below documents the decisions, actions and reviews carried out during both sprints and integrates lessons learned into future work.

Sprint 3 Risk Management Plan

1. Platform & Schema Foundations (Time-boxed DB Decision Spike)

We conducted a **database decision spike** within the first week of the sprint. The goal was to compare relational (Microsoft SQL Server) vs. document-based (MongoDB) data stores for our enrollment and quiz modules. The team analysed concurrency control mechanisms and how they would handle simultaneous enrollment requests and wait-list admissions. SQL Server's row-level locking and lock compatibility matrix were key factors; the Microsoft documentation explains that lock compatibility controls whether multiple transactions can acquire locks on the same resource at the same time and that an exclusive lock blocks all other locks on the same row until it is released. Row-level locks combined with `SERIALIZABLE` transactions also prevent phantom inserts, which is critical for fair wait-list ordering. Based on these findings we chose **SQL Server** as our primary data store because it provides strong ACID properties, supports row-level locks and can enforce transactional invariants needed for concurrent enrollment.

Two formal **Architecture Decision Records (ADR)** were produced as part of the spike:

ADR 1 – Quiz Length and Storage

- **Issue:** Should quiz answers be stored long-term or purged after scoring? Early prototypes stored answers in clear XML files which risked exposure of personal data. Encryption best practices from data-security checklists stress that sensitive data must be encrypted at rest and in transit. Netwrix also notes that personally identifiable information (PII) should always be encrypted, backed up securely and audited regularly.
- **Decision:** We adopted a policy of storing only aggregate quiz scores (e.g., risk categories) and minimal metadata. Raw answers are encrypted at rest with AES-256 and removed after 30 days. During transit, answers are submitted over HTTPS using TLS 1.2. Access tokens restrict viewing to authorised admins.
- **Consequences:** This approach reduces the attack surface for PII exposure and aligns with regulations. It required adding encryption libraries and designing a job to purge old quiz data, but these were accepted as necessary overhead.

ADR 2 – Enrollment and Wait-list Logic

- **Issue:** How should concurrent enrollment requests be handled to avoid over-booking and maintain fairness? Without proper locking, race conditions could allow more students than seats. The SQL Server guide explains that a requested lock can be granted only if it is compatible with the existing lock and otherwise waits until the existing lock is released. Key-range locks ensure that concurrent inserts in a range cannot cause phantoms.
- **Decision:** We implemented a stored procedure that wraps enrollment in a `SERIALIZABLE` transaction with row-level locks on the event and seat counts. The procedure decrements available seats, inserts the student record, performs a post-commit invariant check, and if the seat count is negative triggers a rollback with a retry/backoff loop. Wait-list entries are admitted strictly in FIFO order.
- **Consequences:** This design eliminates double-booking; concurrency tests confirmed that simultaneous enrollment attempts are serialized. The complexity of stored procedures increases but ensures fairness and compliance with capacity rules.

2. Development Environment Standardisation

To ensure team members can clone and run the project within 15 minutes, we paired throughout Sprint 3 to standardise the local ASP.NET environment. According to an article on pair programming, working in pairs encourages continuous feedback and immediate error detection. We leveraged this practice to cross-check each other's setups and build a repeatable checklist. The **Run-It checklist** included:

Step	Description
Clone	`git clone` the repository and check out the `develop` branch.
Install prerequisites	Install .NET SDK 8.0, Node.js 18, and SQL Server Express. Use the provided `install.ps1` script to automate tool installation.
Configure secrets	Copy `appsettings.Development.json.example` to `appsettings.Development.json` and populate database connection string and encryption keys.
Restore & build	Run `dotnet restore` and `dotnet build` from the solution root. NPM dependencies are restored in the `ClientApp` folder with `npm install`.
Seed DB	Execute `./scripts/seed.ps1` to create schema and seed sample data.
Run	Launch the API with `dotnet run --project src/Api`. In another terminal start the SPA with `npm start` (or rely on the integrated spa service). Verify startup by visiting `https://localhost:5001/health`.

The checklist was tested on two fresh laptops (Windows 11 and macOS) and each completed in ~12 minutes. Issues discovered (such as differing `.NET` PATH environment variables) were documented and resolved during the pairing sessions.

3. CI/CD Guardrails (Smoke Build & Health Probe)

We extended our GitHub Actions pipeline to enforce quality gates on every pull request (PR). The pipeline added two jobs: **smoke-build** and **health-probe**.

- **smoke-build:** Runs on `ubuntu-latest` and executes `dotnet restore` and `dotnet build` to ensure the solution compiles. It caches NuGet packages for speed.
- **health-probe:** After a successful build, the API is launched with `dotnet test-server` in the background. A script repeatedly calls the `/health` endpoint until it responds or times out. The `testfully` guide notes that an API health check endpoint should return a 200 OK status when healthy and can monitor downstream dependencies like database connections. If the probe returns a non-200 code or fails to connect, the job fails and blocks the PR. Logs include response codes and times for quick diagnosis.

A sample GitHub Actions YAML used:

```
name: CI
on: [pull_request]

jobs:
  smoke-build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '8.0.x'
      - name: Restore and Build
        run: |
          dotnet restore
          dotnet build --no-restore
  health-probe:
    needs: smoke-build
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '8.0.x'
      - name: Launch API
        run: |
          dotnet publish -c Release -o out
          nohup dotnet out/Api.dll &
          sleep 15
      - name: Probe health
        run: |
          for i in {1..10}; do
            STATUS=$(curl -k -s -o /dev/null -w '%{http_code}'
https://localhost:5001/health || true)
            echo "Health status: $STATUS"
            if [ "$STATUS" = "200" ]; then exit 0; fi
            sleep 5
          done
          echo "Health check failed" && exit 1
```

This pipeline caught broken builds and failing health checks early in the sprint, preventing unstable code from merging into develop.

4. Weekly Risk Reviews and Outcomes

We held **two formal risk review meetings** during Sprint 3 to monitor foreseen and emergent risks.

4.1 Foreseen Risks and Mitigations

Risk	Observation	Mitigation & Outcome
Overbooking due to race conditions (technical/architecture)	As anticipated, concurrent enrollment logic proved tricky. Without locking, simultaneous enrollment requests could decrement seat counts incorrectly, leading to overbooking.	During the spike, we implemented row-level locks and a post-commit invariant check in a stored procedure. When testing uncovered a race condition, we applied these fixes: wrap operations in `SERIALIZABLE` transactions; acquire row locks; perform invariant checks; and retry with exponential backoff. These mitigations eliminated overbooking.
Privacy of quiz responses (security/privacy)	Early prototypes stored quiz responses in plaintext XML. This risked PII exposure.	We encrypted data in transit and at rest using TLS and AES 256. We minimized PII, implemented scoped access tokens and retention policies as recommended by best-practice checklists. The encryption layer successfully protected sample data, and penetration testing found no leaks.
Lost filter state on navigation (UX/usability)	Using WebForms, we observed that filter settings (e.g., sorting or search terms) were lost on page reloads.	We implemented URL query parameters to persist filter state and used a central store to remember scroll positions. This maintained user context when navigating back to lists and improved usability.

4.2 Unexpected Risks and Mitigations

Unexpected Issue	Impact	Response
Misinterpreted pre-assessment quiz requirements	Our initial design treated the quiz as a knowledge test instead of a life-context intake. This mismatch produced irrelevant recommendations and undermined user trust.	We held a requirements workshop with the sponsor, rebuilt the question-to-module mapping spreadsheet to capture life-context questions (e.g., “Are you dating?”), and implemented branching logic accordingly. This realigned the quiz with user needs.
Skills gap in web.config security hardening	A developer was unfamiliar with secure ASP.NET configuration, risking vulnerabilities.	We ran a focused enablement session and adopted a hardened baseline template: enforced HTTPS, configured authentication and roles, set secure cookies and security headers, and blocked direct access to data files. We added security checks to QA and paired on the first hardened commit. As a result, configuration errors were reduced and security improved.

5. Summary of Sprint 3 Lessons

- Choosing SQL Server provided robust locking semantics, preventing overbooking through row-level and key-range locks.
- Encrypting sensitive data at rest and in transit is essential for protecting PII.
- Pair programming improved code quality and knowledge transfer, making it ideal for standardising the development environment.
- Automated CI/CD guardrails with a health check endpoint catch issues early; health endpoints should check dependencies and return 200 on success.

Sprint 4 Risk Management Plan

1. Weekly Risk Reviews and Foreseen Risks

Sprint 4 extended the system to support helpers (mentors) who run cyber-fair sessions. We continued weekly risk reviews to monitor anticipated and new risks.

Foreseen Risk	Status & Mitigation
Helper workspace clutter (technical/UX)	We worried the new workspace might slow helpers due to clutter. To mitigate this, we designed the UI with a minimalist layout and clear sections (Next Up, Prep and Schedule) using progressive disclosure. Progressive disclosure reduces cognitive load by revealing more complex information as the user progresses. Early usability checks showed faster scanning and reduced confusion.
Admin mis-configuration of helper eligibility (organizational)	Incorrectly set eligibility rules could cause unqualified helpers to join sessions. We added presets and input validation (no non-existent tags or expired rules) and implemented a “Who qualifies?” preview before publishing. This allowed admins to check their settings, keeping the impact medium.
Wait-list admissions fairness (technical)	Admitting the wrong person from the wait-list would be high impact. We enforced strict server-side queue ordering, rechecking capacity at admit time and ensuring the UI triggers only server actions. This preserved fairness.
Inconsistent statuses between helper pages (technical)	With multiple helper workspace components (cards, schedule rows, check-in states), status could drift. We ensured both workspace and schedule components read from a single API and event stream, providing a single source of truth.

2. Unexpected Risks and Responses

Unexpected Issue	Impact	Response
Complex ASP.NET code base	The existing code consisted of large code-behind files and over 10 XML data schemas. Business rules were scattered across Page_Load/PostBack branches, user controls and utility classes. This complexity made it hard to extend the system without breaking existing flows (e.g., seat release timers).	We allocated time for code-base familiarisation and mapping. We created diagrams of page lifecycles and data flows, migrated repeated logic into services, and added unit tests for critical methods. This refactoring allowed us to introduce helper workflows (eligibility tags, assignment surfaces, check-in/undo, live status) with fewer regressions.
Complexity of generating test scripts	Creating reliable test scripts for helper workflows took longer because of many moving parts.	We built helper-specific test fixtures that set up sample data, mock roles/eligibility and simulate check-in, seat release and live updates. This investment paid off by enabling automated regression testing.

3. Time-boxed ADRs in Sprint 4

ADR 3 – Organising the Helper Screen

- **Issue:** How should the helper screen be organised? The new helper workspace needed to show certifications, schedules, check-ins and live statuses. A single monolithic view risked clutter and slower scanning. Progressive disclosure advocates breaking complex tasks into manageable steps and revealing information gradually to reduce cognitive load.
- **Alternatives:**
- **Single-page with tabs** – Place all information on one page using collapsible panels.
- **Multi-view layout** – Split the workspace into separate views for certifications, schedule and session management, with a navigation bar.
- **Decision:** We chose the **multi-view layout**. Each view focuses on one category (e.g., Certifications, Schedule, Sessions). Progressive disclosure is used within each view (e.g., accordions show session details on demand).
- **Reasoning:** This design reduces clutter, improves scan speed and allows helpers to focus on one task at a time. Tabs would have overloaded a single page and risked losing context.
- **Consequences:** Requires routing logic and additional components but improves usability and maintainability. Feedback from our first usability test was positive.

ADR 4 – Assigning Helpers to Sessions

- **Issue:** Should helpers be assigned automatically based on eligibility or manually by admins? Auto-assignment could reduce admin work but might not account for context (e.g., a helper's availability or preference). Manual assignment provides control but requires effort.
- **Alternatives:**
- **Automatic assignment** – The system matches helpers to sessions based on eligibility tags and availability.
- **Admin-driven assignment** – Administrators select helpers from a list of eligible candidates.
- **Hybrid** – Automatic suggestions with admin confirmation.
- **Decision:** We selected **admin-driven assignment** for university deployments.
- **Reasoning:** Universities often have specific preferences (e.g., matching a helper who knows the local curriculum). Manual assignment ensures fairness and accountability. The hybrid model was deferred due to complexity.
- **Consequences:** Admins need to perform assignments, but we mitigated workload by providing filters and recommendations. We plan to revisit automation in future sprints.

4. Summary of Sprint 4 Lessons

- **Minimalist, multi-view UI** using progressive disclosure improved helper workflows and reduced cognitive load.
- **Clear eligibility validation and previews** prevented mis-configured requirements.
- **Centralised status updates** ensured consistency across helper pages.
- **Refactoring old code** and investing in test fixtures were necessary due to the complexity of the ASP.NET code base.
- **Manual helper assignment** was chosen over automation to accommodate university-specific preferences.

Conclusion and Next Steps

Across Sprints 3 and 4, the team strengthened the project's technical foundations, mitigated privacy and concurrency risks, and delivered new helper management functionality. Key takeaways include the importance of choosing a database with strong locking semantics, encrypting sensitive data, leveraging pair programming for environment standardization and implementing automated CI/CD guardrails with health checks. Looking ahead, we will continue refining the helper workflows, explore hybrid assignment mechanisms and expand automated testing to cover more edge cases. Regular risk reviews remain a cornerstone of our process, ensuring that both technical and organizational risks are surfaced early and addressed promptly.