Architecture-Driven Development

University of
St Andrews

170002815 · 170007256 · 180014200 · 190004947

Word Count:

CS5033 · Practical 2

22 March 2022

1 Overview

This report explores an example architecture for a healthcare system which handles the management of patient information and appointments. The process of creating the architecture is explored in Sections 2 and 3, with the resulting architecture being described in Section 4; Sections 5 and 6 provide an implementation plan for and a critical evaluation of the chosen architecture respectively. The report is completed by a conclusion (§7) and set of references.

word count

2 Architecting Process

2.1 Brief Descriptions

rename subsect

The process that was utilised to derive this architecture was not particularly complex but was one that required a substantial amount of detailed analysis and critical thought. There were two primary phases during which the conception of the presented architecture took place, which were the Pre-Planning Phase, and the Discussion and Planning Phase.

Pre-Planning Phase

To begin, the architects for this software convened, in order to discuss the first steps and identify the basic requirements and fundamentals. From the outset, it was clear that there were a number of considerations that were required to be taken into account. These considerations were the cornerstones in the architectural design process and were crucial in informing the final architecture.

Primarily, one of the most important considerations was the security of the system. Health data is sensitive for a number of reasons. The leakage of sensitive information or any security breaches could put the individuals whose health information was infringed, in an extremely vulnerable position. Individuals could lose their jobs, support networks, housing, etc. or be at risk of psychological risk and/or social ostracization if certain information became public knowledge. Furthermore, the release of such sensitive information could facilitate attempts to commit identity fraud. (https://www.ncbi.nlm.nih.gov/books/NBK9579/.)

do cite

Keeping this in mind, it was of utmost importance that the system was able to store data, in accordance with the HIPAA (Health Insurance Portability and Accountability Act) guidelines. This Act was extremely useful in guiding the architectural development of this system, as it illustrated very stringent guidelines about how healthcare providers could use patient data, what could be disclosed and to whom. Alongside this, the HIPAA guidelines were also very clear on how data should be handled, maintained and transmitted, specifically regarding the technical safeguards required, such as ensuring patient data is encrypted during rest and transit, audit controls for all hardware and software management and integrity controls to ensure ePHI (electronically protected health information). Hence, this knowledge informed the expectations of the architectural framework developed for this scenario and was embedded in the final architecture.

Another extremely critical consideration was the robustness of the system. One of the key assumptions of healthcare systems is that they must be functional most of the time. This is because medical emergencies can happen at any time, and it is essential for the system to be able to handle requests, such as extracting patient records or viewing recent treatments, at any given time. For instance, in the event that a new medication has had an immediate onset of severe symptoms, the Ambulance Service Crew must be able to view the patient records and the new administered treatment, in order to make an informed guess about the potential issue, and then treat the issue. Hence, if the system is down, this information would not be accessible, which could have potentially fatal consequences. As such, it was extremely critical to ensure that the system would continue to function, despite the existence of faults in its constituent subsystems. Thus, it was fundamental that the architectural design followed some semblance of a component-based architecture, or a fallback plan, in order to ensure that the system could function in the presence of extenuating contextual factors.

The final essential consideration was the performance of the system. As medical professionals require patient data in real-time, it is essential that the system is able to respond to requests within a specific frame of time. Hence, considerations such as latency (time spent responding to an event) or channel capacity (the number of concurrent events) are essential for the development of this architecture.

Thus, once these two principles were identified, they underpinned the rest of our architectural derivation process and the team chose to conduct independent research, in order to identify the most suitable solutions.

Planning Phase

Once the independent research phase had been concluded, the team reconvened, in order to compare solutions and develop the most fitting solution. Each team member put forth an initial proposal of an architectural framework that would befit either a specific component of the system or the overall system as a whole. This allowed the team to discuss and refine the presented proposals, and incorporate distinct aspects of the solutions into the final system.

The team elected to meet in the John Honey Teaching Labs (JHB: JH110), as the use of a whiteboard was required for the purposes of designing the architecture.

The discussion was prefaced with a group read-through of the specification provided, during which, the key stakeholders and actors were identified, such as the staff in the General Practice (Nurse, General Practitioner, Administrator), the members of the Ambulance Service (Crew and Administrator), the staff in the Hospital (Administrator and Doctor) and finally, the patients. This was represented in a whiteboard sketch.

Once these key stakeholders were identified, their associated tasks and behaviours were associated with them. For example, the General Practitioner would be expected to make and attend digital appointments, order test kits, etc, while the Ambulance Service Administrator would be expected to log incoming calls and dispatch assistance to those who have requested so. As the expected functionality would enable the team to design the most effective architectural frameworks for the constituent subsystems, this step was essential.

Following this, the team discussed the assumptions that were made during the process of deciding which architectural framework would fit best. These assumptions are presented in a more detailed manner in the above sections. However, prior to the codification of such assumptions, the team discussed and debated each premise, in order to ensure that our model most accurately represented the real-world behaviours of the General Practice/Hospital/Ambulance Service/patient stakeholders. Like the expected functionality, it was key to develop the assumptions at this stage as they would go on to play a major role in informing the final architecture plan that was designed. This was added to the whiteboard sketch, before delving into the architectural framework development.

Finally, once the actors, their expected functionality and assumptions had been decided upon, the team discussed some fundamental architectural styles and patterns that could be utilised to represent the system. Some of the styles and patterns discussed were a Publish-Subscribe system for the Ambulance Service Crew and Health Boards, a potential Event Queue system for the General Practitioner, etc.

The initial whiteboard sketch that represents all of these elements and potential solutions is illustrated below.

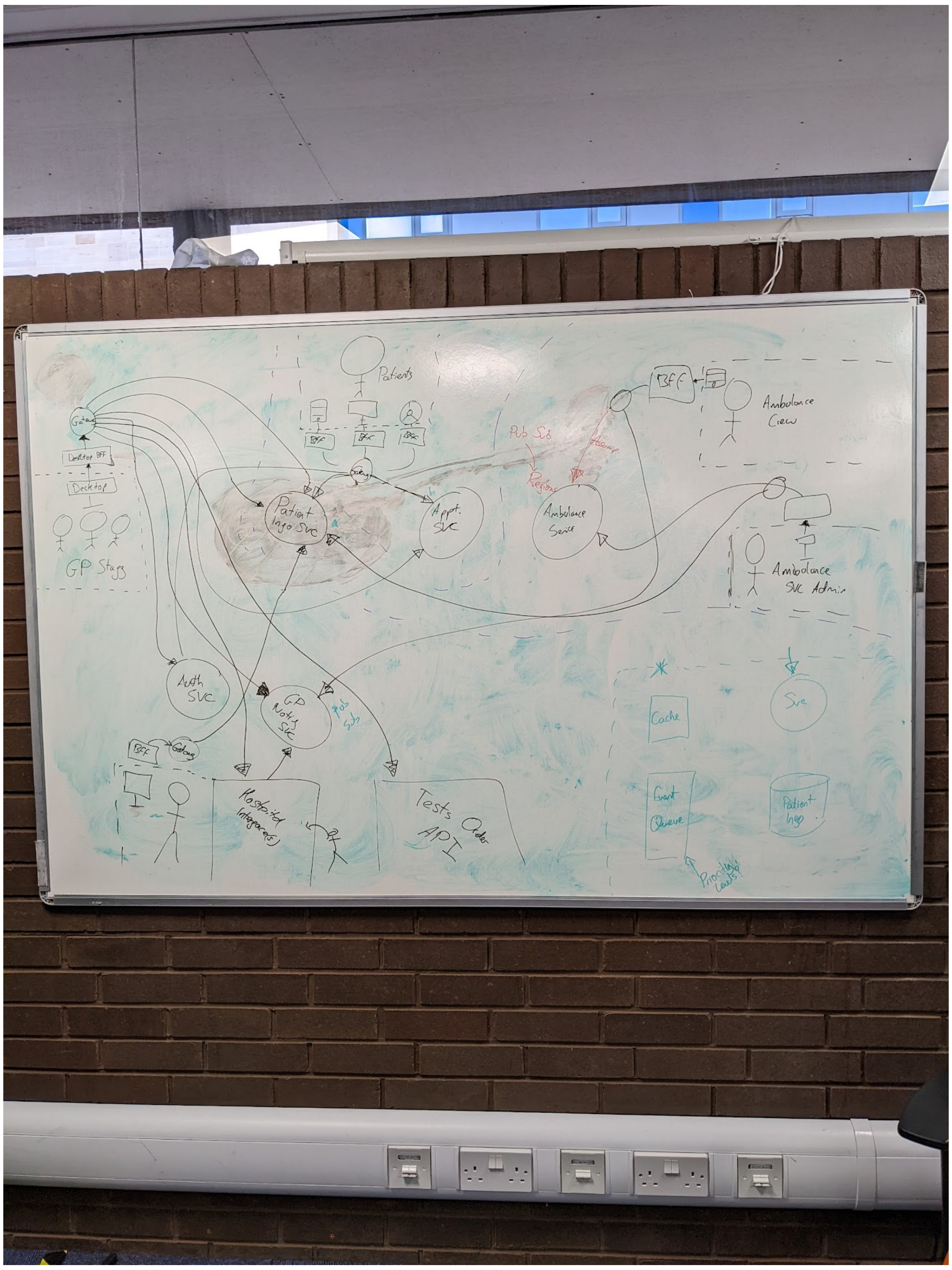


Figure 1: Initial whiteboard sketch of the proposed architecture.

As seen above in Figure 3, the stakeholders of the system are illustrated, such as the General Practice staff, Health Board staff, patients and Ambulance Service staff, with some simple associated services that are represented in circles. The different entities that interact with these activities are represented through arrows, which makes it easier to conceptualise the varying access permissions and data requirements that each Actor has.

The bottom right corner represents an example sketch of the architecture discussed for the General Practitioner. The notion of a cache is used to address the real-time requirements of the General Practice when accessing patient information. The patient information is represented as being accessed through a database (as noted from the diagram), and the event-queue system is a basic solution to represent the list of tasks that must be completed by the Practice. Despite only sketching out an architecture for one subsystem, this exercise proved to be immensely helpful in designing the rest of the architecture. Specifically, this enabled the team to critically evaluate the specific functionalities, data management procedures and architectural characteristics of different entities of the system, and consider the concrete implementation strategies that needed to be reflected within the architecture. Thus, this sketching method was highly effective in guiding the team's design process, as it highlighted the specific requirements and intersystem behaviours that needed to be considered.

Once every parameter of the system had been considered, the team was able to design a full system architecture for the specified scenario, which capitalised on a large part of the discussion that took place during the planning phase.

The final architectural specification is discussed in detail in the next section.

2.2 Resources

The primary literature resources used to influence the design, evaluation, and implementation plan of the chosen architecture were Robert C. Martin's Clean Architecture [1], and Mark Richards and Neal Ford's Fundamentals of Software Architecture [2].

Clean Architecture provides information on architecture from a practical standpoint, including fundamental principles to follow when architecting a system, including a set of fundamental component principles. The principles relating to the architecture of components which are explored Clean Architecture are divided into two groups: those which deal with component cohesion, and those which deal with component coupling. The principles relating to component cohesion are as follows.

- **Reuse/Release Equivalence Principle (REP)**:
- **Common Closure Principle (CCP)**:
- **Common Reuse Principle (CRP)**

The principles relating to component coupling in Clean Architecture are as follows.

- **Acyclic Dependencies Principle (ADP)**:
- **Stable Dependencies Principle (SDP)**:
- **Stable Abstractions Principle (SAP)**:

2.3 Technologies

Throughout the design process, the following technologies were used.

- **Microsoft Teams** [3]: Used for group text and video communication.
- **Google Drive** [4]: Used to store and share images, progress reports, meeting notes, and other documents.
- **LaTeX** [5]: Used for document formatting.
- **GitHub** [6]: Used to store LaTeX code and enable asynchronous work on the report by different team members.
- **Lucidchart** [7]: Used to create architecture diagrams.
- **diagrams.net** [8]: Used to create system functionality flow charts.

3 Requirements Analysis

3.1 Functional Requirements

-The system shall be able to create and facilitate digital appointments.

-The system shall allow patients to create digital appointments.

-The system shall allow General Practitioners to create digital appointments.

-The system shall be able to securely store sensitive patient data.

-The system shall be able to authenticate patients that interact with the application, by verifying their details against data obtained from the General Practitioners' records.

-The system shall be able to authenticate General Practitioners, General Practice Administrators, Nurses, Ambulance Service Administrators and Health Board Administrators that interact with the application, by verifying their details against data obtained from the National Medical Professionals Database.

-The system shall only allow patients to be referred to the General Practice, following a digital appointment, except in emergencies.

-The system shall allow the patient records to be updated by the General Practitioner.

-The system shall allow the General Practitioner to refer the patient to the Local Health Board if the required service is available locally.

-The system shall only allow the General Practitioner to refer the patient to the National Health Board if the required system is not available locally.

-The system shall allow the patient details to be updated by their General Practitioner, following a digital appointment.

-The system shall allow the patient records to be updated by the General Practitioner, to indicate that a digital appointment has taken place.

-The system shall allow the patient records to be updated by the Health Board Administrator, when the patient goes to A&E in a Hospital, to indicate that treatment of some kind has taken place.

-The system shall allow Hospital doctors to view patient details and vitals.

-The system shall allow Hospital doctors to add entries on diagnoses made.

-The system shall allow Hospital doctors to add entries on any treatment given.

-The system shall notify the General Practitioners when the Ambulance Service Administrator or Health Board Administrator has updated the patient details.

-The system shall allow the General Practitioners to create digital appointments for patients.

-The system shall allow the General Practitioner to view patients' vitals, such as blood pressure, heart rate, heart-rate variability, oxygen saturation and respiration rate.

-The system shall allow patients to view their vital data that is stored within the system.

-The system shall allow patients to view any changes or new entries made to their vital data, as a result of their digital appointment.

-The system shall allow General Practitioners to access the full record of any patient.

-The system shall allow General Practitioners to order tests for a specific patient that is affiliated with the Practice.

-The system shall allow Nurses to view details of patients' treatments, tests or test results.

-The system shall not allow Nurses to view patients' sensitive information or vitals.

-The system shall allow General Practice Administrators to view appointment details between patients and General Practitioners.

-The system shall allow General Practice Administrators to make or delete appointments, on the behalf of patients.

-The system shall allow General Practice Administrators to obtain a general overview of the Practice's day-to-day activities.

-The system shall allow General Practice Administrators to create statistical reports of the Practice's performance, such as the number of appointments per month, the number of test kits ordered, etc.

-The system shall not allow General Practice Administrators to view any patient information.

-The system shall allow Ambulance Service Administrators to log calls in real-time.

-The system shall allow the Ambulance Service Administrators to view the specific regions where an ambulance is available.

-The system shall allow the Ambulance Service Administrators to dispatch an ambulance that is closest to the region where the call was logged.

-The system shall allow the Ambulance Service Administrators to view patient records and details.

-The system shall allow the patient records to be updated by the Ambulance Service Administrator, to indicate that treatment of some kind has taken place.

-The system should allow Ambulance Service Crew to indicate when each call-out has been addressed.

-The system shall shut down, in the event of a potential cyber threat.

3.2 Non-Functional Requirements

- -The system shall be able to handle all users that attempt to access, update or modify patient details.

-The system shall be able to handle all users that attempt to access or update treatments given.

-The system shall be able to handle all users that attempt to access or update diagnoses made.

-The system shall be able to support 3,000 concurrent users.

-The system shall be able to process all identifiable data under the Data Protection Act, 1998.

-The system shall be able to obtain real-time patient vitals in 0.1 seconds.

-The system shall be able to load all operations within 1 second.

-The system shall be able to process all requests within 2 seconds.

-The system shall be available between 24 hours a day, 7 days a week, with the exception of a maintenance window.

-The system should have centralised logs that can maintain all services, instances and possible errors in a single location.

-The system shall be accessible via mobile devices and personal computers.

-The system shall be compatible with the latest product-release OS versions.

-The system shall be able to interface with Binah.AI, in order to enable video-calling and obtain real-time patient vitals.

3.3 Organisational Requirements

-Users of this system shall identify themselves using their National Medical Professionals' Database credentials.

-The system shall be down for maintenance, on the last working day of each calendar month, for 30 minutes.

-The system shall be evaluated for response times on the last working day of each calendar month.

-The system shall be able to prevent cross-scripting attacks.

-The system will not store unencrypted sensitive data.

-Database security must meet the Health Insurance Portability and Accountability Act (HIPAA) requirements.

-All patient data will be retained in system archives for up to 5 years.

3.4 External Requirements

-The system should be able to support an annual growth of 10 General Practices that can utilise this platform.

- The system should be able to support a 100% growth in user concurrency, and still meet all defined functional and non-functional requirements.

- The system should utilise cloud-based solutions that facilitate scalability.

3.5 Assumptions

For the purposes of this practical, there were a number of assumptions that were made, in order to streamline the process of designing a system software for General Practices and to consolidate the expectations of this platform.

These assumptions and their justifications, are listed as follows.

1. **We assume that a Health Board Administrator is responsible for redirecting a referred patient to the nearest Hospital that offers their required service.** For example, a student based in St. Andrews would have their patient details and records stored within the Fife Health Board. This is the Health Board that the local General Practice (example- Blackfriars or Pipeland) would refer the patient to. A Health Board could encompass multiple Hospitals - for example, the Fife Health Board could include Adamson Hospital (Cupar), Victoria Hospital (Kirkcaldy), St Andrews Community Hospital, etc. Thus, when a patient is "referred" to a local Health Board, we assume that the General Practice does not refer the patient to a specific Hospital. Hence, we assume that a Health Board Administrator is responsible for selecting the specific Hospital that the patient will be referred to, and as such, will pass the patient's details (vitals, treatment records, diagnoses, personal information, etc) onto this specific hospital. This will ensure that the patient's details are communicated on a need-to-know basis, thereby reducing the chances of sensitive information being released to inappropriate individuals.

2. **We assume that the Ambulance Service Administrator is able to dispatch ambulances that are nearest to the individual calling for assistance.** This indicates that the Ambulance Service Administrator retains a degree of autonomy when dispatching ambulances, in order to ensure that assistance is provided as quickly as possible. An example of a scenario where this would be useful would be a call requiring assistance to be sent to St Andrews. If there are two ambulances in Kirkcaldy and Cupar, we would want the Ambulance Service Administrator to be able to send the ambulance in Cupar, as it is closer. Hence, we do not expect the system to be automatically dispatching ambulances, as the calls come in. However, we do expect the system to be able to provide a recommendation of which ambulance is closest (computed using third-party navigation software), which would help the Ambulance Service Administrator dispatch assistance.

3. **We assume different members of the General Practice/Health Board/Ambulance Crew have different access permissions, regarding patient data.** Primarily, we assume that General Practitioners have access to all patient information. This includes the patient vitals (heart-rate, heart-rate variability, oxygen saturation, respiration rate, parasympathetic activity, etc), sensitive patient information (date of birth, gender, height, weight, etc), patient diagnoses and treatment. We assume that Nurses only have access to patient diagnoses and treatment. We assume that General Practice Administrators have access to no patient data. We assume that these individuals can only view anonymised appointment data. We assume that Health Board Administrators have access to no patient data, except the information that the General Practitioner has sent over, regarding the services required by the patient. We assume that the Hospital doctors have read-only

access to patients' vitals and sensitive data, but can update patient diagnoses and treatment. We assume that the Ambulance Service Administrators have no access to patient data, except the data communicated by the patient during the ambulance call. We assume that the Ambulance Service Crew have read-only access to patient vitals and sensitive patient information, but can update patient diagnoses and treatment.

4. **We assume that the patient does need to make a digital appointment when being seen by a Nurse/General Practitioner in person.** We assume that the patient must make an appointment by phoning the General Practice, in order to be seen by a Nurse or General Practitioner in real life. This allows us to model real-world behaviour as closely as possible, as we would expect that General Practices require an appointment to be made, in advance. This holds true whether it is a digital appointment or an in-person appointment. We also assume that the steps following an in-person appointment are the same as those that follow a digital appointment. Specifically, the patient's vitals are taken and added to the database, treatment is offered/the patient is referred and test kits are ordered.

5. **We assume that all medical personnel records are registered in a comprehensive database - the National Medical Professionals' Database.** We assume that the credentials utilised to authenticate users are obtained from a centralised database that contains the information of all registered medical professionals within the country. This allows the system architecture to ensure that only authorised medical personnel are allowed to access patient records and that any externally created user accounts (such as system administrators) are automatically unauthorised to view sensitive information.

4 Architecture

4.1 Modelling Language

A relatively informal modelling language is used to describe the final architecture produced for this report, with the C4 model being used in combination with a notation similar to UML. This combination was chosen as all team members were relatively familiar with the concepts used for both practices, focussing the project's time to designing and evaluating the architecture, rather than learning modelling languages.

C4 Model

The C4 model primarily consists of four different architectural viewpoints, described in the form of diagrams, which represent an architecture as a hierarchical set of abstractions [9]. The viewpoints are designed to reflect to how software architects and developers think about and build software, and are summarised as follows.

- **System Context Diagram**: A viewpoint which describes a system's architecture as a whole, allowing viewers to see the "big picture". The diagram has a focus on the users of the system, rather than specific details, such as which technologies are used. Section 4.2 explores the system context diagram for the chosen architecture in this report.
- **Container Diagram**: A viewpoint which shows the details of a single subsystem within a system context diagram. The diagram is composed of containers, which represent an application or datastore (e.g. a web application, filesystem, database, etc.), and high-level references to other subsystems. Section **???** explores several container diagrams for the chosen architecture in this report.
- **Component Diagram**: A viewpoint which describes a single container within a container diagram, with respect to the major building blocks (i.e. components) that make up the container. Component diagrams contain some specific details, such as the technology used to implement or communicated between services, but still provide a relatively high-level overview of a container. Section **???** explores some component diagrams for the chosen architecture in this report.
- **Code Diagram**: A viewpoint which describes a single component within a component diagram as it is implemented within code, including the classes and interface involved, and their relationships. Code diagrams are not explored in this report, as they reference relatively low-level implementation details, and could be automatically generated from code written during the system's development.

The C4 model also includes three supplementary diagrams: the system landscape diagram, the dynamic diagram, and the deployment diagram. However, none of these are reference within this report, as the four core diagrams are sufficient to capture the chosen architecture [9].

code dia-
grams no
used

only som
comp di-
agrams
shown fo
berevity
as the
system is
very larg

find out
from Ben
& Jorda
what no-
tation is
used

Notation

4.2 Overview

The architecture derived for the scenario listed in Appendix A is fundamentally a service-oriented architecture (SOA), as the key functionality of the system is implemented by multiple independent, distributed services. The core services used and their roles within the architecture are as follows.

- **Appointment Service**: Handles create, read, update, and delete (CRUD) operations relating to appointments, employing business rules in combination with authentication to authorise information appropriately. This service is described further in Section 4.3.
- **Patient Information Service**: Handles CRUD operations relating to patient information, employing business rules in combination with authentication to authorise information appropriately. . This service is described further in Section **??**.
- **Ambulance Service**: Affords the logging of calls and the dispatching of ambulances. This service is described further in Section 4.6.
- **GP Notification Service**: Provides the ability to asynchronously pass notifications to GPs. This service is described further in Section 4.5.

which
reqs are
satisfied
by which
svc?

In order to enable user interaction with services, a client-server-like architectural style is employed for user-service interactions; Section 4.7 discusses this further.

Figure 2 displays the system context diagram (see §4.1), where the set of core services, users, and interactions can be seen.

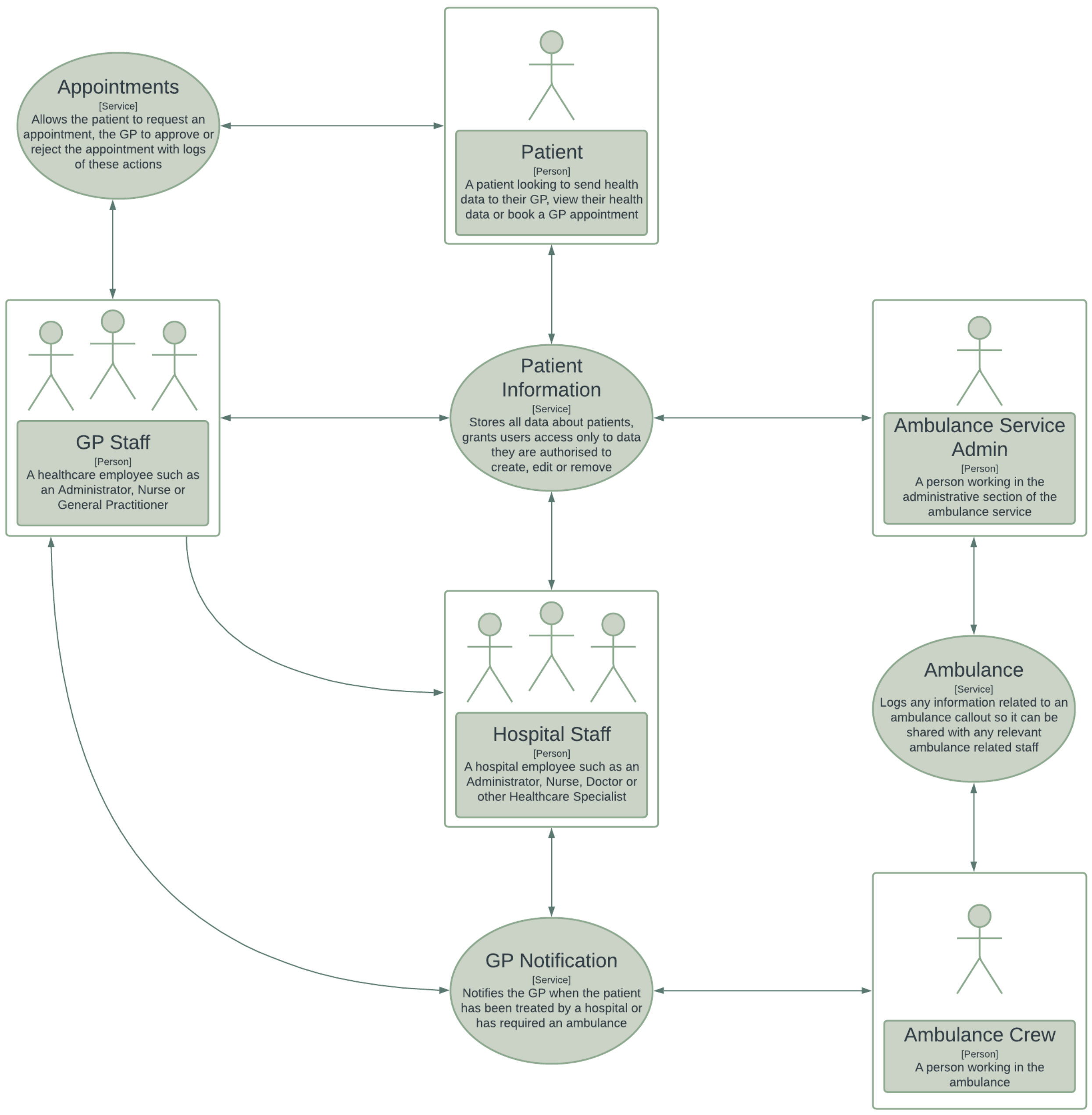

Appointments
[Service]
Allows the patient to request an appointment, the GP to approve or reject the appointment with logs of these actions
Patient
[Person]
A patient looking to send health data to their GP, view their health data or book a GP appointment
GP Staff
[Person]
A healthcare employee such as an Administrator, Nurse or General Practitioner
Patient Information
[Service]
Stores all data about patients, grants users access only to data they are authorised to create, edit or remove
Ambulance Service Admin
[Person]
A person working in the administrative section of the ambulance service
Hospital Staff
[Person]
A hospital employee such as an Administrator, Nurse, Doctor or other Healthcare Specialist
Ambulance
[Service]
Logs any information related to an ambulance callout so it can be shared with any relevant ambulance related staff
GP Notification
[Service]
Notifies the GP when the patient has been treated by a hospital or has required an ambulance
Ambulance Crew
[Person]
A person working in the ambulance

Figure 2: The system context diagram for the chosen architecture, displaying the service-oriented approach used as the fundamental architecture for the system.

Service-Oriented Architecture

As the target system's domains can be partitioned with relative ease, an SOA is a suitable choice for the overarching architectural style [10]. Although there are other applicable architectural styles for the target system, the following benefits of an SOA were the primary driving factors in the decision to prioritise it over other architectural styles.

- Services can be developed independently, reducing time-to-market, as independent teams can develop each service, and improving business agility, as each service's business rules can be adapted independently, provided the boundaries within the architecture are enforced effectively (see §)[11]. This is primarily an advantage over more rigid architectures, such as those using a monolithic style.

ref SOA not an arch in Eval sec

- Services can be deployed independently, facilitating the maintainability, short-term scalability (elasticity), and long-term scalability of the target system. This is primarily an advantage over architectures which are designed to have a one-to-one mapping from system instance to deployment machine, such as an architecture employing a layered monolithic style.
- Services are more independent with respect to failure than other architectural styles. This means that if one service fails, the failure of other services is not guaranteed, which is particularly beneficial for the target system, as the domains are mostly disconnected. For example, if the GP Notification Service fails, the other services in the system are not guaranteed to fail, meaning that the system will still provide the majority of its functionality to users. Overall, this makes the system more resilient to

- Similarly to failure, the services are more independent with respect to performance than other architectural styles. For example, if the GP Notification Service handles requests slowly due to a bug in its implementation, the Ambulance Service would be unaffected in terms of performance, provided it is deployed on a different machine; this is particularly beneficial for the target system, as some services, such as the Patient Information Service, require a certain level of performance to function as required.
- Due to the network-based nature of an SOA, services and clients are typically designed to be more robust to failure, improving the overall reliability of the system.
- Integration with external systems, such as the , is facilitated, as the target system will be deployed on a network due to the nature of an SOA. Similarly, integration with legacy systems which might need to be included in the newly developed system is facilitated [11].

hospital system

- The enforcement of architectural boundaries of the target system's domain is improved, as each service's boundary can be mapped directly to each domain's architectural boundary [10].
- The domain-driven separation of services, in combination with their low levels of interaction, mitigate the negative effects of an SOA, including selecting an appropriate granularity and handling service choreography/orchestration [10].

4.3 Appointments Service

The Appointments Service handles read and write requests relating to patient appointments. The service is designed with a relatively typical architecture for a read-heavy workload (see ???), consisting of a service which contains the business logic for the Appointments Service, a datastore, and a cache. Figure 3 displays the container diagram for the Appointments Service.

assump that ap-point-ments ar read mor than writ ten

right dia-gram?

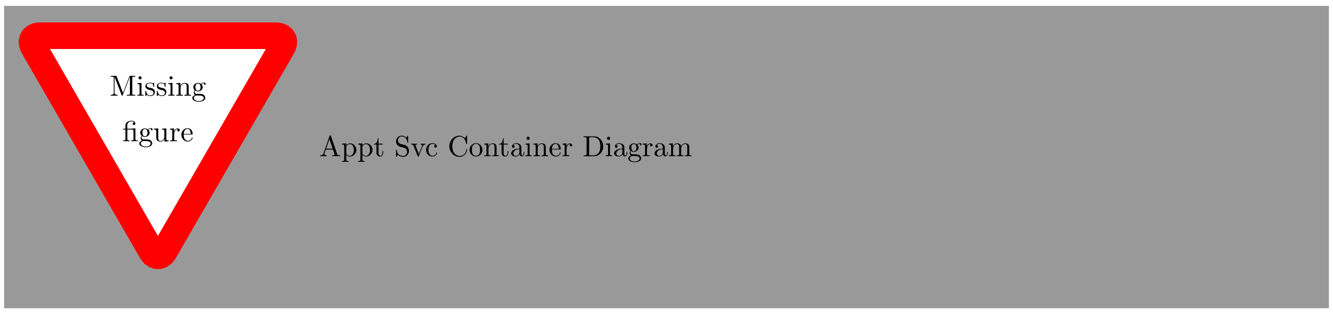


Figure 3: The Appointments Service described using a component diagram.

Datastore

The Appointments Service's datastore is used to persist data relating to appointments, improving the service's reliability. From Figure 3 (CAN IT?), it can be seen that the datastore is integrated within the architecture as a dependent on the service component within the Appointments Service, rather than the other way round. This integration style employs the Stable Abstractions Principle (see §2.2), as the dependency is pointing from the less stable datastore to the more stable business contained within the service component.

can it?

The application of the SAP also follows the advice given in Martin's Clean Code regarding how datastores should be integrated within architectures [12], and also does not confine the choice of data storage technology, allowing for solutions ranging from a simple file system to a complex RDBMS to be used, improving both the maintainability and scalability of the Appointments Service.

Cache

To improve the performance of the Appointments Service with relatively low overhead, an in-memory cache is used to respond to some requests without querying the datastore. In order to do so, the cache would store requests and their respective results as they are returned from the service/datastore after they had been retrieved from the service/datastore for the first time

Two key considerations when introducing a cache into a system are how consistency and coherency will be handled during its operation. Handling these concerns, particularly if the cache or service to which the cache is applied to is replicated, can quickly become a complex problem which introduces a significant amount of overhead [13]. Fortunately, as the Appointments Service's workload is read-heavy (see ???), the overhead introduced is likely worth handling when considering the potential performance gains introduced by the cache. Other overhead introduced by the cache, including storing entries, checking for entries, and handling evictions is considered to be marginal when compared against the potential performance gains provided.

complete ref to assumption

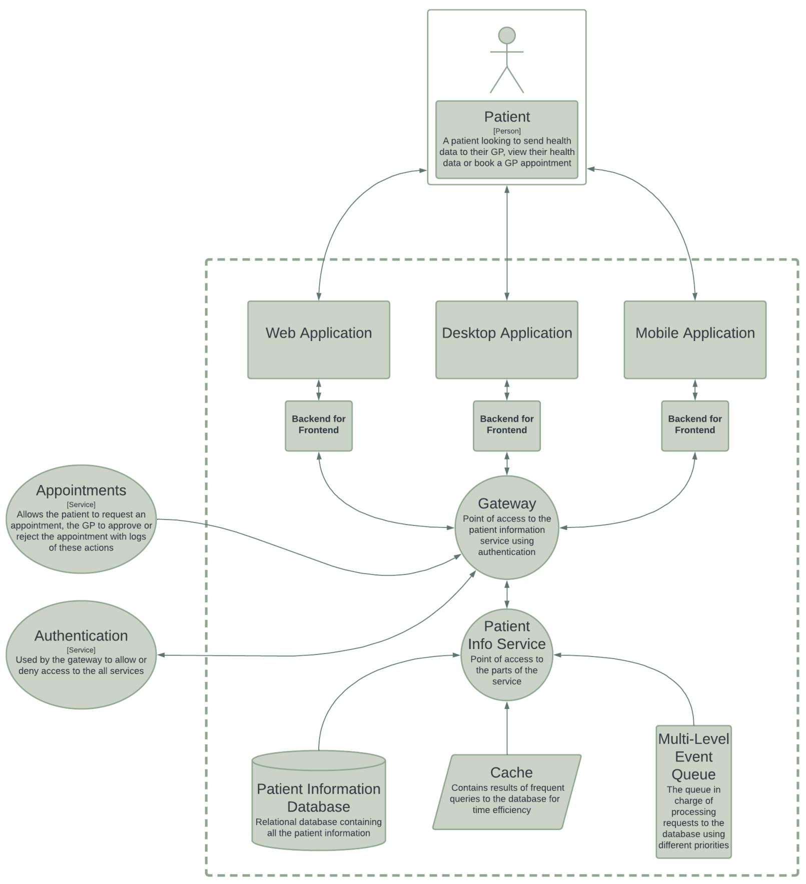
As it is assumed that there are many users , it is likely that evicting items from the cache on a commonly used lease-recently-used (LRU) basis is inefficient. This is because the cache will quickly fill up with requests relating to individual users which cannot be reused for other users (e.g. fetching a user's set of appointments), potentially evicting cached information which can be reused (e.g. the grand total of number of appointments for all users). While an alternative cache eviction policy, such as least-frequently used (LFU) or re-reference interval prediction (RRIP) policies, it considered as an implementation detail to determine this, and is therefore excluded from the scope of the architecture.

cite assumption

4.4 Patient Information Service

The Patient Information Service handles all create, read, update, and delete (CRUD) operations relating to patient information, including personal information, such as patient addresses, as well as medical information, such as blood pressure readings. The service employs the use of role-based authentication to provide different users access to different data, as information handled by the service is considered highly sensitive [14] and different users need different access permissions as per the system requirements

cite assump


Patient
[Person]
A patient looking to send health data to their GP, view their health data or book a GP appointment
Web Application
Desktop Application
Mobile Application
Backend for Frontend
Backend for Frontend
Backend for Frontend
Appointments
[Service]
Allows the patient to request an appointment, the GP to approve or reject the appointment with logs of these actions
Gateway
Point of access to the patient information service using authentication
Authentication
[Service]
Used by the gateway to allow or deny access to the all services
Patient Info Service
Point of access to the parts of the service
Patient Information Database
Relational database containing all the patient information
Cache
Contains results of frequent queries to the database for time efficiency
Multi-Level Event Queue
The queue in charge of processing requests to the database using different priorities

Missing figure

Patient Information Container Diagram

Figure 4: The Appointments Service described using a component diagram.

Datastore & Cache

As the data handled by the Patient Information Service and the Appointments Service (see §4.3) are both related to individual patients and are associated with read-heavy workloads (see ???), the datastore and

ref assumption

cache used within the Patient Information Service are similar to those used in the Appointments Service (see §4.3), with respects to both operation and reasons for introducing them as part of the architecture (i.e. reliability and performance).

Multilevel Priority Queue

One way in which the Patient Information Service differs from the Appointments Service is the requirement to handle bursty write behaviour when patient information is generated by GPs, as described by ??? . As this behaviour contrasts significantly to typical read-heavy workloads of the Patient Information Service, an additional architectural feature of a multilevel priority queue is introduced to handle all write requests sent to the service.

ref assumption

A multilevel priority queue is a task queue which consists of multiple queues, each of which associated with a different priority level. Inputted tasks are enqueued onto a queue with respect to their given priority level, with the processing element(s) (e.g. the service component in the example architecture) taking tasks from the data structure based on descending priority, meaning that more important tasks are completed before less important ones [15].

A multilevel priority queue is employed within proposed architecture to support the unique bursty write workloads without interrupting the standard non-bursty write workloads. Bursty write workloads, such as patient information gathered during testing procedures, are prioritised lower than non-bursty workloads, such as standard updates to patient information (e.g. a change in address), with the service component handling tasks in an order with respect to their priority. The queue nature of the multilevel priority queue also enables for write requests to be delayed as to avoid disrupting and invalidating read workloads, which are likely more important.

Although multilevel priority queues are commonly implemented as multilevel feedback queues, where tasks are moved between queues depending on the amount of time they have been worked on for [16], such an implementation is unsuitable for the target system's architecture, as it is likely unfair to change the priority of each task once it has been assigned.

4.5 GP Notification Service

The GP Notification Service is a comparatively simple service which provides the ability to asynchronously pass notifications to GPs through the use of a publish-subscribe messaging system. This service was introduced in the architecture for the following primary reasons.

1. The GP Notification Service allows for the GP client device to be decoupled from the services which send notifications to GPs, such as the .

   which service?

2. The addition of future features which require GP notifications are greatly facilitated, as the new features would simply need to interact with the GP Notification Service, with no other architectural changes being required.
3. Future scaling of the system is better supported with a publish-subscribe architecture in comparison to direct communication between services sending notifications and GP interface devices.
4. The GP Notification Service can buffer notifications sent to GPs, enabling asynchronous communication, which in turn can improve the reliability of the system. For example, if a GPs device briefly goes offline, notifications are not necessarily lost as the GP Notification Service can buffer them while the device is offline; however, if services were to communicate directly with GP devices, then notifications would have likely been lost in this scenario.

Figure 5 displays a container diagram for the GP Notification Service.

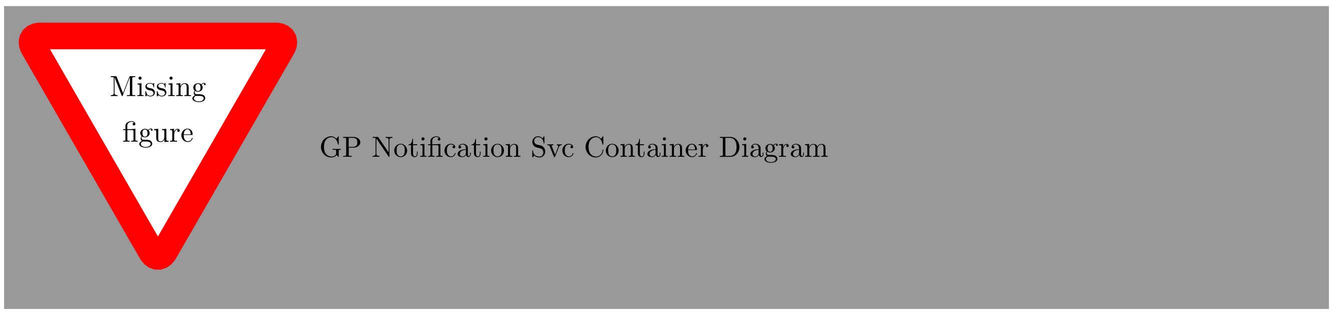


Figure 5: The Appointments Service described using a component diagram.

Publish-Subscribe Pattern

The GP Notification Service applies an event-driven architectural style in the form of a publish-subscribe (pub-sub) ,system which buffers notifications sent from other services, such as the , what?

A pull configuration is used, where the GP client device receives notifications by polling the GP Notification Service, rather than a push configuration, where the GP Notification Service would actively communicate with the GP client device to deliver notifications, for the following reasons.

1. The GP devices are external to the system, meaning that initialising communication from the GP Notification Service to each GP device is complicated to achieve in comparison to the other way around [17].
2. The rate of communication (e.g. polling frequency) can be dynamically adjusted based on the GP device's current network connection. This is more beneficial than the GP Notification Service having this control, as the GP device's network connection is likely to fluctuate more than the GP Notification Service's connection.
3. The polling interval can be adjusted for each individual GP by the GP's device, allowing for a trade-off in network usage and the amount of delay caused by the gap between the notification being received and polled. For example, if the GP's device has a poor internet connection or is battery powered, the rate of polling could be decreased to improve network efficiency or reduce power consumption.
4. The notifications being sent to GPs are assumed to not be time-critical . ref as-sumption

4.6 Ambulance Service

The Ambulance Service handles all ambulance-related concerns for the target system, including the logging of calls, the dispatching of ambulances, and the indication of callouts being dealt with. The ambulance-related requirements of the target system's domain appears relatively disconnected from other requirements, providing good reason for this service to be a part of the chosen architecture for this report.

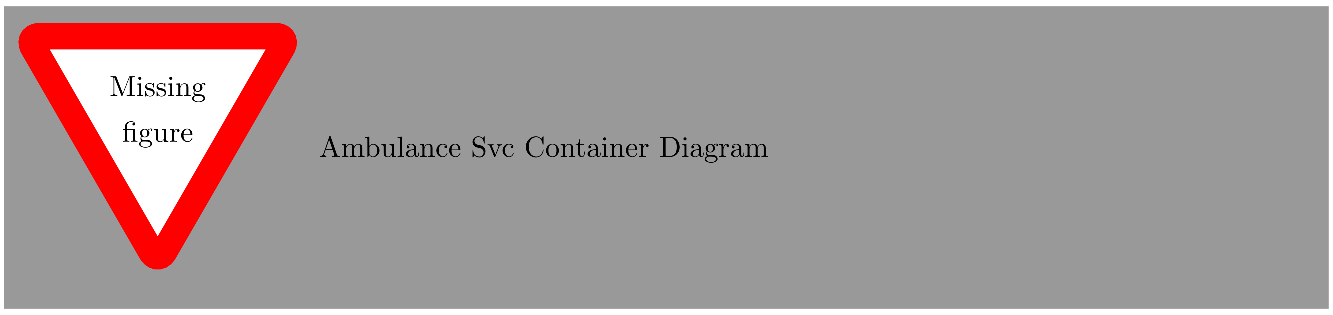


Figure 6: The Ambulance Service described using a component diagram.

Publish-Subscribe Pattern

how is th pubsub used?

Similarly to the GP Notification Service (see §4.5), the Ambulance Service employs an event-driven architectural style in the form of a publish-subscribe (pub-sub) messaging system. The pub-sub system is used primarily for performance purposes, and therefore uses a push configuration for message delivery, contrasting to the pull configuration used for the GP Notification Service.

A push configuration is more performant in comparison to a pull configuration with respect to message delivery latency, because the delay between the message arriving at the Ambulance Service and being sent to the client device is marginal, in comparison to a pull configuration where the delay is on average half of the polling interval [17]. Although a push configuration adds complexity from communicating with client devices outside the system's internal network, and reduces flexibility for the power and network consumption for client devices, these trade-offs are accepted in favour of the reduced message delivery delay, as this characteristic is critical to the Ambulance Service's functionality.

4.7 Clients

The clients within the architecture offer the functionality of the services in the form of a user interface (UI). Figure ??? displays WHAT . what

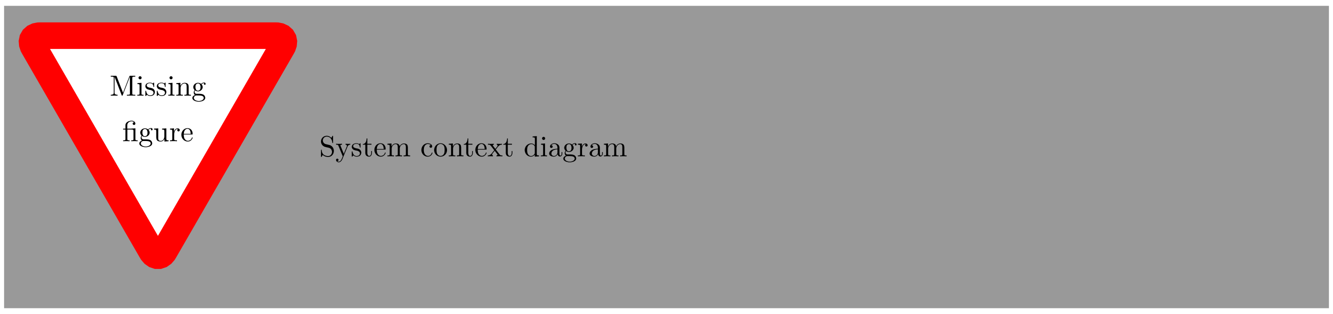


Figure 7: A container diagram for the WHAT? client, displaying the use of the gateway and BFF architectural patterns.

The overarching architectural style applied for communication between clients and services is the client-server style, where the clients, such as patient mobile devices, communicate with internal services, such as the Patient Information Service, through the use of standard web technologies (e.g. HTTP/HTTPS) to offload computation and data-storage responsibilities. In order to provide seamless, flexible, reliable, and secure integration with the services, the clients exhibit two primary architectural patterns within the client-server style: gateways and backends for frontends.

Gateway Pattern

The gateway architectural pattern employs an intermediate service to mediate requests which pass from client devices to the services within a system, rather than having client devices communicate directly with the services. This can be seen in Figure ???, where the . finish sen

The gateway architectural pattern provides the following advantages.

- Cross-cutting concerns, such as authentication, and data transformations, can be handled in a uniform manner for all client devices, simplifying the implementation of each user interface [18].
- The communication protocol used by requests can be changed before being forwarded to services, allowing for client devices to communicate using web-based protocols (e.g. HTTP/HTTPS) with services that are implemented to communicate using non-web-based protocols, such as QUIC [19][18].
- As all requests are passed through a single location before entering the internal system where the services lie, the gateway pattern facilitates the application of performance and security measures, such as load balancing and request filtering.
- Requests to different services can be coalesced to reduce data ingress and egress for client devices, improving performance. For example, if a patient's mobile device was loading the patient's information and appointments, the two separate requests to the Patient Information Service and Appointments Service could be combined into a single request to the respective gateway node, reducing the required network communication overhead [18].

The primary disadvantages of the gateway architectural pattern is the increased architectural complexity and the potential performance concerns surrounding the gateway services becoming a bottleneck or single point of failure. As the gateways used in the chosen architecture in this report have potential to be stateless, they can be replicated to mitigate the issues of performance and reliability; however, the issue of increased architectural complexity cannot be mitigated. Overall, the advantages of the using gateways for the target system's architecture outweigh the disadvantages.

Backend For Frontend Pattern

The backend for frontend (BFF) architectural pattern is the application of one intermediary service (the backends) per client-side user interface (the frontends).

While similar to the gateway pattern, and often considered synonymous [20], the BFF pattern is applied within the chosen architecture for this report for different reasons to the gateway pattern, which are as follows.

- The development of frontend applications using the humble object pattern is greatly facilitated, as the business logic for the applications are contained within the backend services instead. The humble object pattern is the separation of logic from frontend components and increases the testability of such components; the application of the humble object pattern is generally considered to be good software-engineering practice [21].
- Each BFF backend can be tailored specifically to its respective frontend, allowing for platform-specific intricacies to be handled cleanly, in turn facilitating development, and improving performance and maintainability. This is in contrast to the gateway pattern which uses a single backend for all frontends.

The trade-offs of using both the BFF and gateway pattern in the target system's architecture are discussed in Section ??? of this report.

4.8 Authentication

Authentication is a critical part of the system's functionality, as all services require some form of authentication to function in-line with the system's specification. For example, the .

In order to include authentication within the system's architecture, the OAuth 2 framework [22] was chosen for the following reasons.

- It is a proven authentication framework, as it is used by renowned companies such as Facebook, GitHub, and DigitalOcean [23]
- It enables single-sign on for the target system, which is beneficial from a user standpoint [24].
- The protocol used to authenticate users is simple in comparison to other authorisation techniques.
- It provides simple integration with third-party services [23]. This is beneficial for the target system, as external systems, such as those used within hospitals, may require authorisation for their required access to the system.
- The primary architectural elements required for the framework can be "bolted" onto the rest of the architecture, allowing for the component coupling dependencies to be followed (see §2.2), in turn improving the flexibility and maintainability of the architecture as a whole.

5 Implementation

5.1 Implementation Plan

Following our C4 model, the implementation of the patient digital appointments and management system should take a top-down approach. This should begin with the system context, then the containers, then the components, followed by the actual classes. By structuring the implementation in the same order that the architecture was designed, the architecture-driven development falls into a natural order. Top-down implementation, also known as 'stepwise refinement', imposes a hierarchical structure on the program matching that of the C4 architecture. By starting with defining the solution at the highest level of functionality, not only does the implementation begin and remain structured but individual development teams can be assigned to specific branches of the program maintaining architectural boundaries while promoting scaling and performance. These initial branches would be skeletons for the web application, desktop application, and mobile application as well as the initial gateway, database, cache, and event

deploy or different nodes to ensure failures are indep If the target user-base was small, it might be worth using a single node for cost, but the size of th NHS user base is not small queue. It is clear that another advantage of the top-down approach is that the foundational elements of the system are implemented first meaning that they're available once the individual services are developed. Another benefit of organising the development in this manner is that the existing services are built into the system as a primary concern rather than as an after thought. We have inferred from the system specification that the the services that assign patients to hospitals and that order testing kits are both external services. Following a bottom-up implementation paradigm, once the core functionality of the system was built out, code would have to be revised and refactored to integrate with these external services. The main goal of architecture-driven implementation is to minimise architectural drift an erosion. Drift refers to introduction of a design decision in the code that was not included in the architecture but does not necessarily violate its constraints while architectural erosion refers to design decisions in the code that do violate the originally produced architecture [1]. The methodologies to reduce these properties are discussed in the section below.

5.2 Agile Development Paradigm

By using the agile development paradigm and adding to the codebase in sprints, the individual development teams assigned to different services will stop and re-evaluate their progress respective to each other periodically. Not only does this enable precise organisation but it also gives the teams a chance to reflect on the architectural degradation. At the end of each sprint cycle, the implementation is to be 'synced' with the architecture. This could mean updating either the code or the initial architecture depending on the conclusions made during an analysis. In line with the agile manifesto [2], continuously focusing attention on good design decisions will enhance agility. Syncing of the implementation and architecture should not be limited to sprint meetings but should also occur whenever a 'fault' is encountered. In this context, a fault is defined as when it is uncovered that the implementation will not work as intended due to a misalignment of goals with the architecture. If the development team can effectively manage the gap between the architecture and implementation then the documentation will most accurately reflect the implementation allowing other stakeholders to stay informed for decision-making processes. This includes, but is not limited to, allocating development resources to the different parts of the system as a response to varying levels of success across the development teams.

5.3 Enforcement of Architectural Styles and Constraints

To ensure that our implementation meets all the non-functional requirements and to require a high level of security, reliability, and availability while safely managing concurrent processes we must 'hoist' these properties. George Fairbanks defines architectural hoisting as "the direct ownership, management, or guarantee by the architecture of a feature, property, or quality attribute" [3]. As an example, we could employ Enterprise Java Beans (now known as Jakarta Enterprise Beans) to hoist the properties of concurrency and scalability. EJB is a Java API that has various purposes such as managing processes and threads while running concurrency management. Another method which will push our implementation to enforce the architectural styles and constraints is by developing the system in a test-driven manner while deriving the tests from the architecture. At the lowest level, we have unit tests for each smallest element of the architecture. These tests will insist that these 'building blocks' each function correctly on their own before we can check cross-service functionality. After the unit tests, we can also derive integration tests from the architecture. Each connection in the various design diagrams represents an interface between two containers. components, or classes. Integrations tests shall be written to check that each interface behaves appropriately and that the subsystems of the patient digital appointments and management system produce the expected output from various types of input. These tests should not overwhelm development but should be comprehensive enough that 'most' bugs are caught. Of course it is difficult to explicitly define 'most' but the amount of tests written will come down to how the product manager allocates development resources. Furthermore, we could take inspiration from Netflix's ChaosMonkey and test our services after taking different nodes down to simulate various failures. In production we should not expect to have such failures due to an AWS setup as described below but due to the vital nature of the system, it is necessary to be able to provide the main functionality even when part of the system has failed. Another type of testing that can be used to enforce architectural constraints is acceptance tests. Acceptance testing is critical for ironing out faults before deployment, especially in the context of a medical system for which performance and reliability are of utmost importance. Acceptance testing should be run by the individual development teams with access to false datasets to simulate normal system use. A test should be written for each line of interaction through the system to ensure that patients, GPs, the ambulance service, the hospitals, and the health boards are all able to use the services provided by the system correctly. From the flow diagram [4] we can see two different interaction sequences that have two branches created at each decision point. Our acceptance tests should simulate entire interaction sequences and should assert the correct output at each step. This way, we can say with some confidence that the system serves the purpose it is built for.

5.4 Twelve-Factor App

Twelve-factor app is a methodology for software development which consists of 12 principles that should be followed for successful development in the large [5]. When implementing our healthcare system we should be vigilant about each of these principles and should apply them as much as possible. The first of which regards the codebase; the codebase shall be tracked in a version control system such as Git allowing for a high level view of the current implementation while the individual development teams build out their respective services. It also means that in the case of a failure in a production deployment, we could simply rollback the system to a previous version that was working correctly. The other note about the codebase by the twelve-factor app manifesto is that no components should repeat code; all shared code should be factored into libraries which can the injected into the individual components via the dependency/package manager. Furthermore, the application should never rely on the implicit existence of system-wide packages. The codebase should be structures such that all dependencies are declared completely and exactly in a dependency declaration manifest. It is also essential for a dependency isolation tool to be used as well to avoid implicit dependencies leaking into the rest of the system. If these to protocols are followed, new developers are able to deterministically build and run the codebase with only the language runtime and dependency manager installed. The other advantage of this is that architectural boundaries are respected and represented in the code. Another noteworthy principle of the twelve-factor app is the principle on backing services. Since our system, as represented in the architecture, relies on multiple different services working in harmony with one another, the implementation should not make distinctions between local and third-party external services. For example, the code should treat a local MySQL database just as it would an instance of the Amazon Relational Database Service. The result of this is that, without refactoring or code changes, external services can be switched for one another. This corollary of this is that new versions of the individual services can be deployed into the system without the need to change the existing implementation which promotes scaling.

5.5 Technologies

Before discussing a potential technology stack for the patient digital appointments and management system, I would like to give an overview of the deployment and hosting technologies we can leverage to achieve the high levels of availability and reliability that is required of a critical healthcare system such as the one we have designed. The first of which are AWS Elastic Load Balancers (ELBs) which will distribute the system traffic across multiple EC2 instances in multiple availability zones. The AWS EC2 instances are virtual servers in Amazon's elastic compute cloud on which our implementation would be deployed in order to take advantage of the Amazon Web Services infrastructure. Not only will the load balancer share the traffic across the nodes to reduce the chance of failure due to stress but it will also redistribute traffic from an unresponsive node to other nodes in the pool. This means that even in the event of a single server failure, the system can continue to run and provide the essential healthcare services. Another product from the AWS family we would utilise is the CodeDeploy system. By using AWS CodeDeploy we would be able to coordinate application deployment and updates across an arbitrary number of EC2 instances. By automating deployments, following the twelve-factor app principle of disposability, we would be able to quickly push new versions to production without the cost of coordinating deployment protocols. It also asserts that the same application revisions would be deployed across all live environments in a consistent and predictable manner. Furthermore, AWS CodeDeploy would allow us to maintain a high level of availability, reliability, and performance due to its active instance health tracking in accordance to custom rules. This would enable us to avoid downtimes and target high-traffic service routes for performance upgrades. Moving on from cloud computing, the implementation should also utilise the Java framework Spring. Spring is an application development framework for Java which is used to build high-performance, easily testable, and reusable code. Spring supports quick start up and is heavily focused on dependencies which aligns with the twelve-factor app principles for disposability and dependencies. The framework provides a transaction management interface that is highly scalable and supports the requirement for safe concurrency in our system. Another technology that we could leverage is React. React is a JavaScript library for front-end implementations that focuses on reusing components and not repeating code across classes. This property aligns with the twelve-factor principle of keeping shared code within libraries that can be exposed to explicit classes which need them. Using the react library for our front-end, we could also exploit the advantages of the React Native framework which integrates by design with ReactJS. React Native automatically detects the platform that the application is being run on and then generates the correct code for the platform. This would be extremely beneficial when implementing our system across the desktop application for the general practitioners, the application for the hospital systems interface, and the mobile application for patients. Although React is a JavaScript library, it could still be compatible with the proposed Java Spring back-end as they could communicate via JSON which is mutually supported. As a backend service we could deploy an Amazon RDS instance which could be managed alongside the EC2 nodes using the AWS Elastic Beanstalk service. AWS RDS automatically backs up the database while integrating with the other AWS services to all be managed by the Elastic Beanstalk service which can orchestrate deployments and updates.

5.6 Implementation References

- Whiting, E and Andrews, S. (2020). Drift and Erosion in Software Architecture. Proceedings of the 2020 the 4th International Conference on Information Systems and Data Mining. Available at: https://dl.acm.org/doi/pdf/10.1145/3404663.3404665
- Agile Manifesto (2019). Principles behind the Agile Manifesto. Available at: https://agilemanifesto.org/principl
- George Fairbanks. Architectural Hoisting (original blog post). Available at: https://www.georgefairbanks.com/l hoisting-original/
- Ira's system flow diagram
- The Twelve-Factor App. Available at: https://12factor.net

6 Evaluation

6.1 Current Considerations

Architect Biases

- IRA - did a lot of research on styles - natural biases - influence from past project & current trends - SOA is held highly - monolithic layered are shunned (good for critical system)

Data Loss in Event-Driven Styles

The use of event-driven styles within the architecture includes the pub-sub messaging systems employed in the GP Notification Service (see §4.5) and the Ambulance Service (see §4.6), as well as the multilevel priority queue used in the Patient Information Service (see §4.4). Each of these applications of event-driven architecture comes with the risk of data being lost, as the intermediate stores for the respective "events" are memory-based, meaning that data will be lost upon service failure if the correct precautions are not taken.

The proposed architecture takes step to mitigate data loss within the GP Notification Service by backing-up notifications which are buffered within the pub-sub messaging system to a resilient datastore (see §4.5); the Ambulance Service's pub-sub system also has similar functionality. Steps can also be taken with pub-sub systems involving acknowledgement messages (acks) to ensure that messages are only removed from the pub-sub system once they have been correctly handled by the message receiver [17]; however, this concern is not facilitated by the architecture and instead relies on developer vigilance to be implemented correctly.

However, the multilevel priority queue within the Patient Information Service does not currently have any architectural support for providing fault-tolerance with respect to write requests which arrive in the queue. Improvements to the architecture could include the use of a simple append-only log-file to reduce the amount of data lost in a fault, and more complex fault-tolerance mechanics could be applied to mitigate almost all data loss; however, the overhead introduced by such a scheme could be counterintuitive as much patient data (e.g. blood pressure readings) could likely be regenerated fairly easily.

Authentication Performance Bottleneck

The support for authentication provided by the architecture offers many benefits, as discussed in Section 4.8. However, as all client devices must communicate with the Authentication Service in order to access the internal services' functionality (see §4.8), there is a significant risk that the Authentication Service will become a bottleneck for the system.

Solutions to this problem could include replicating the Authentication Service, with replicas either set to handle a specific user type (e.g.) or being distributed geographically to reduce latency for user's connection with the Authentication Service. It should be noted that replicating the service could incur

example
user type

significant overhead to the system, meaning that replication should only be used if absolutely required [25].

BFF & Gateway Patterns

As discussed in Section 4.7, both the backend-for-frontend (BFF) pattern and gateway pattern are applied within the system's client architectures. Many systems will either employ one pattern or the other, but rarely both together, as the benefits they provide overlap significantly; in fact, the two patterns are sometimes considered synonymous [20]. The use of both patterns introduces one extra level of indirection to an extra service node, increasing complexity and incurring a network overhead, both of which could be avoided by using only one of the two patterns.

However, as discussed in Section 4.7, while the gateway and BFF patterns provide some shared functionality, they also provide functionality unique to each pattern which would be more difficult to apply with the use of only a single pattern.

A potential improvement to this area of the architecture is removing the gateway nodes from the architecture and including their functionality within the BFF nodes through, with the use of the sidecar pattern being employed to provide the shared functionality reuse provided by the gateway pattern without incurring the extra overhead [26].

Size

- IRA

- Splitting into teams could be Tricky - Overarching architects or not? (trade-offs)

Data Security

One key consideration for the target system is that of data security, as patient health data can be considered to be very sensitive information [14]. In evaluating the proposed architecture, it can be seen that the architecture supports the implementation of data security in the following ways.

- All sensitive data is stored in a single datastore within the Patient Information Service (see §4.4). This means that the introduction and management of security policies and measures is greatly facilitated during the system's implementation and lifecycle.
- The integration of the OAuth 2 framework within the architecture facilitates the secure authentication of users using a modern protocol (see §4.8).

6.2 Lifecycle Considerations

System Evolution

- IRA - Arch is flexible & maintainable w.r.t. future changes - Application of component principles helps with this - Documentation

Architectural Drift & Erosion

- IRA

- Documentation

- From lack of dev vigilance: - stick with arch - uphold boundaries - write docs - remove cruft (cite Martin Fowler) - team - some hoisting has been applied (e.g. service separation), but still left to developers - SOA is not **that** important in an arch, but instead we should focus on boundaries (see Clean Architectrue book ch 27 [27])

ATAM

- IRA - If possible?

7 Conclusion

Overall, this report provides an overview of an architecture for the system described in Appendix A, along with a detailed breakdown of the system's requirements, an implementation plan, and a critical evaluation of the proposed architecture.

A range of functional and non-functional requirements were derived from the

A wide variety of both common and unusual architectural features, patterns, and styles have been applied to shape the proposed architecture towards the needs of the target system, including the use of the simple client-server pattern (see §4.7) and a multilevel priority queue (see §4.4).

References

[1] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson, 2017.

[2] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly, 2020.

[3] "Microsoft Teams," Accessed: 2022-03-25. [Online]. Available: https://www.microsoft.com/en-gb/microsoft-teams/group-chat-software

[4] "Cloud Storage for Works and Home, Microsoft Teams," Accessed: 2022-03-25. [Online]. Available: https://www.google.com/intl/en-GB/drive/

[5] "LaTeX - A document preparation system," Accessed: 2022-03-25. [Online]. Available: https://www.latex-project.org/

[6] "GitHub," Accessed: 2022-03-25. [Online]. Available: https://github.com/

[7] "Lucidchart," Accessed: 2022-03-27. [Online]. Available: https://www.lucidchart.com/

[8] "Diagram Software and Flowchart Maker, diagrams.net," Accessed: 2022-03-27. [Online]. Available: https://www.diagrams.net/

[9] "The C4 model for visualising software architecture," Accessed: 2022-03-23. [Online]. Available: https://c4model.com/

[10] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly, 2020, ch. 13, pp. 163–177.

[11] "SOA (Service-Oriented Architecture), IBM," Accessed: 2022-03-25. [Online]. Available: https://www.ibm.com/uk-en/cloud/learn/soa

[12] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson, 2017, ch. 30, pp. 277 – 283.

[13] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* O'Reilly, 2021, ch. 9, pp. 321–383.

[14] "Health data in the workplace, European Data Protection Supervisor," Accessed: 2022-03-27. [Online]. Available: https://edps.europa.eu/data-protection/data-protection/reference-library/health-data-workplace_en

[15] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating Systems Concepts.* Wiley, 2014, ch. 5.3.5, pp. 166 – 167.

[16] ——, *Operating Systems Concepts.* Wiley, 2014, ch. 5.3.6, pp. 168 – 169.

[17] "Choose a subscription type, Cloud Pub/Sub, Google Cloud," Accessed: 2022-03-26. [Online]. Available: https://cloud.google.com/pubsub/docs/subscriber

[18] “The API gateway pattern versus the Direct client-to-microservice communication, Microsoft,” Accessed: 2022-03-25. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern

[19] “QUIC, a multiplexed transport over UDP,” Accessed: 2022-03-25. [Online]. Available: https://www.chromium.org/quic/

[20] “Pattern: API Gateway / Backends for Frontends, Microservices.io,” Accessed: 2022-03-25. [Online]. Available: https://microservices.io/patterns/apigateway.html

[21] R. C. Martin, *Clean Architecture: A Craftsman’s Guide to Software Structure and Design.* Pearson, 2017, ch. 23, pp. 211 – 215.

[22] “OAuth 2.0,” Accessed: 2022-03-27. [Online]. Available: https://oauth.net/2/

[23] “An Introduction to OAuth 2, DigitalOcean,” Accessed: 2022-03-23. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2

[24] “Why Your Organization Should be Using OAUTH 2.0, Clowder,” Accessed: 2022-03-27. [Online]. Available: https://www.clowder.com/post/why-your-organization-should-be-using-oauth-2.0

[25] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* O’Reilly, 2021, ch. 8, pp. 273–318.

[26] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach.* O’Reilly, 2020, ch. 17, pp. 245–265.

[27] R. C. Martin, *Clean Architecture: A Craftsman’s Guide to Software Structure and Design.* Pearson, 2017, ch. 27, pp. 239 – 247.

A Scenario

You are required to develop an architecture for a patient digital appointments and management system.

This system should act as a central point for creating and delivering patient appointments as well as storing patient data. Digital appointments should be created via this system, allowing a GP to ascertain and monitor vital patient data. The patient’s own data should also be stored in the system allowing for a full 360 degree view of patient information. This system is being built in order to alleviate pressure on the health service.

Each person resident in Scotland is registered with a General Practitioner (GP) practice. The health care service will maintain a record of each person’s name, a unique health care id, date of birth, contact details, information on next of kin, the GP practice they are registered with and medical history. Patient records contain highly sensitive information. A patient may be seen via digital appointments or at the practice by a nurse (for minor injuries or tests) or a GP. A patient can only access the GP practice if they are referred to after a digital appointment, except in case of emergencies when they can either call an ambulance or go directly to the accident and emergency (A&E) department of a hospital. An ambulance crew will either treat a patient on site or take them to a hospital. A GP will either treat a patient themselves or refer the patient to a hospital run by the local health board, except when the required service is not available locally. Each appointment or treatment will result in an entry in the person’s medical history, which should be accessible across different health boards and services. When a patient has been treated at a hospital or by an ambulance crew, a notification should be sent to their GP to flag up the new entry.

The GP should be able to generate digital appointments for patients. These appointments should allow the GP (via the patient’s personal device) to obtain vital information about the patient – namely their blood pressure, heart rate, heart rate variability, oxygen saturation and respiration rate. This data should be generated in real-time and stored securely in the patient information system. The system should provide different functionalities depending on the category of user. A patient must be able to use the service to make appointments at their GP practice. They should also be able to see all the vital data generated by all their digital appointments in the system. GPs should be able to access the full record of any patient without delay, initiate appointments and add entries to the patient record. They should also be able to order one or more tests for the patient within the practice or refer the patient to a hospital, either within or outside the local health board. Nurses can see a limited part of the patient record and add entries relating to treatment, tests or test results. Practice administrators can make or cancel appointments for patients and produce statistical reports on the performance of the practice without accessing details of any patient.

An ambulance service administrator must be able to log calls to the service and dispatch ambulances to patients who require them. This aspect of the service is highly time sensitive. They can also update patients' records according to the service delivered. Ambulance crews indicate when each callout has been dealt with. Hospital doctors can view all the details of any patient they see and add entries on diagnoses made and treatment given.

A well-designed and implemented system should also support the following features:

- An intuitive UI appropriate to the user category,
- Access authentication for different categories of users and restriction of available information and functionality accordingly,
- Support for multiple types of devices,
- Concurrent access,
- Support for several different views and analyses over the data,
- Validation of input data where applicable,
- Generation of digital appointments and using patients' devices functionalities to extract the necessary patient data, and
- Deal with potential uncertainties when patients' devices cannot extract this data.