Architecture-Driven Development

University of St Andrews

170002815 · 170007256 · 180014200 · 190004947

Word Count: 9180

CS5033 · Practical 2

22 March 2022

Contents

1 Overview

This report explores an example architecture for a healthcare system designed to handle 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.

2 Architecting Process

2.1 Architecting Phases

The process that was used to derive this architecture was consisted of two primary phases, were the Planning Phase and Design Phase.

Planning Phase

The first phase to take place was the Planning Phase, which focussed on determining the requirements which the architecture should support and the characteristics which the architecture should display. The determined requirements are listed in Section 3, and the characteristics which were highlighted during discussion are as follows.

1. **Security**: One of the most important characteristics for the architecture to support is the security of the system. Health data is sensitive for a number of reasons, with its leakage potentially putting the individuals whose health information was infringed in vulnerable position [1]. Both the NHS' documentation regarding data security [2] and other resources, such as the HIPAA guidelines [3][4], were consulted during the architecture's design in order to cater for this concern.
2. **Robustness**: The robustness of the system is a critical consideration, as healthcare systems must be operations for 24 hours a day, 7 days a week. This is because medical emergencies can happen at any time, and it is essential for relating systems to be able to handle requests, such as extracting patient records or viewing recent treatments, at any given time. As such, it was determined that the system should continue to function, despite any faults within its constituent subsystems, leading to a component-based architecture being used.
3. **Performance**: The final essential consideration was the performance of the system. As medical professionals require the reading and writing of 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 and channel capacity were considered essential during the development of the proposed architecture.

Design Phase

The design of the architecture took place iteratively, with initial architectural designs being completed by different group members, and later being combined into a final architecture which was repeatedly refined.

All designs are centred around the key stakeholders and actors of the system, including GP staff, Ambulance Service staff, Hospital staff, and patients, as well as their associated tasks and behaviours. For example, architectures would consider the GPs and how their tasks of making and attending digital appointments, order test kits, etc. could be supported by the architecture. Considering the required functionality while designing the architecture enabled the discovery of the most effective architectural frameworks for the constituent subsystems.

Throughout the design process, several ambiguities within the scenario were discovered, leading to a set of assumptions to be made for the final architectural design; these assumptions are presented in Section 3.3. Alongside the consideration of actors and requirements, fundamental architectural styles and patterns that could be utilised within the system were discussed. Discussed styles and patterns include the publish-subscribe pattern for the Ambulance Service Crew, and the event-driven style for the General Practitioner.

Figure 1 displays the final architecture which was determined during the Design Phase, where it can be seen that the stakeholders of the system are illustrated, including as the GP staff, Hospital staff, patients, and Ambulance Service staff, along with the chosen 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 communication required between actors and services.

The bottom right of the diagram in Figure 1 displays a proposal for the architecture of the Patient Information Service. It can be seen that the proposed architecture is similar to that of the final architecture of the service, which is discussed in Section 4.5.

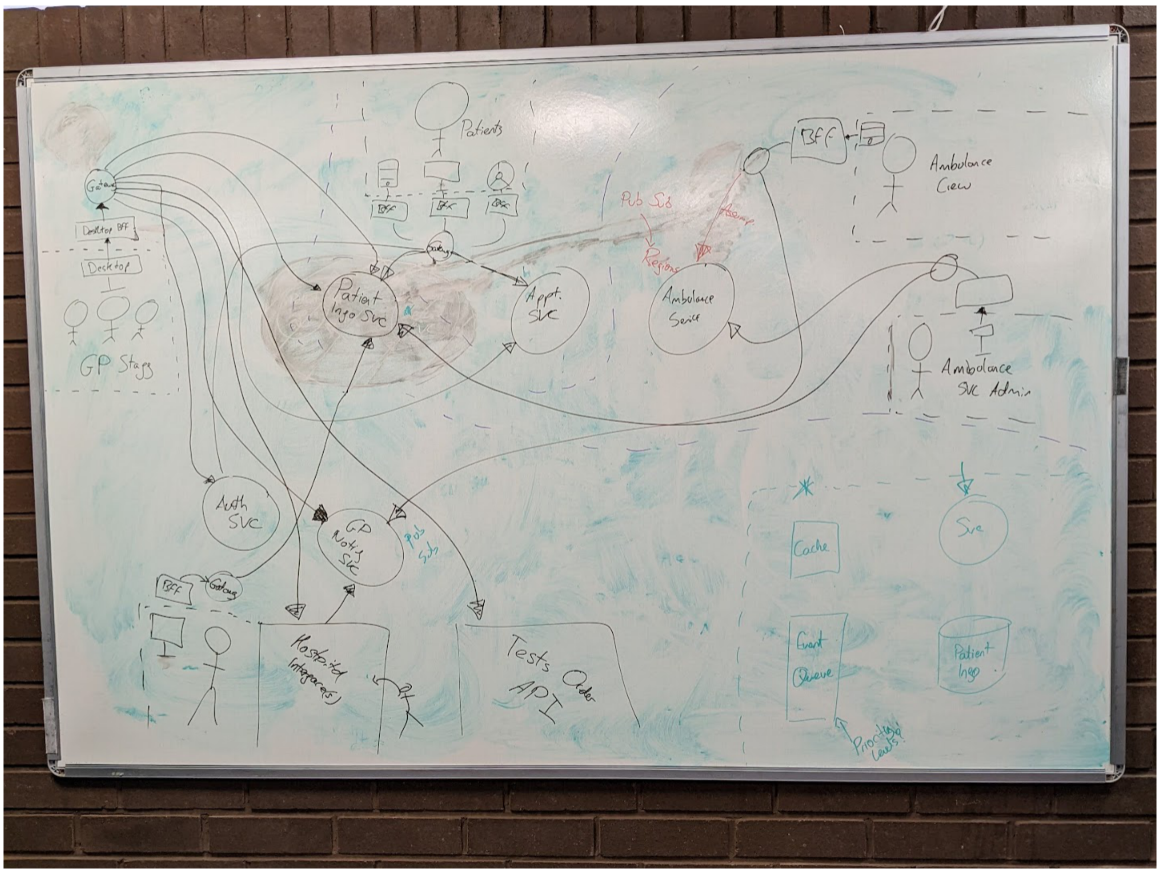

Figure 1: Initial whiteboard sketch of the proposed architecture.

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 [5], and Mark Richards and Neal Ford's Fundamentals of Software Architecture [6].

Fundamentals of Software Architecture was used to influence the choice and application of different architectural styles, patterns, and features. Clean Architecture provides information on architecture from a practical standpoint, including fundamental principles to follow when architecting a system, including a set of component principles, which influenced the design of the proposed architecture. These component principles 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)**: Component constituents which are released together should be reused together and vice versa.
- **Common Closure Principle (CCP)**: Component constituents which change for the same reason and at the same time should be placed in the same component.
- **Common Reuse Principle (CRP)**: Components should not contain excess constituents in a way which would force users to depend on things they do not need.

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

- **Acyclic Dependencies Principle (ADP)**: The component dependency graph should not contain any cycles.

- **Stable Dependencies Principle (SDP)**: Dependencies between components should point in the direction of stability, leading to most dependencies pointing in the direction of the system's business rules.
- **Stable Abstractions Principle (SAP)**: Components should be as abstract as they are stable. For example, a stable component (e.g. a business rule implementation) should be implemented as interfaces and abstract classes, while a volatile component (e.g. a web interface) should try to avoid such abstractions.

2.3 Technologies

Throughout the design process, the following technologies were used.

- **Microsoft Teams** [7]: Used for group text and video communication.
- **Google Drive** [8]: Used to store and share images, progress reports, meeting notes, and other documents.
- **LaTeX** [9]: Used for document formatting.
- **GitHub** [10]: Used to store LaTeX code and enable asynchronous work on the report by different team members.
- **Lucidchart** [11]: Used to create architecture diagrams.
- **diagrams.net** [12]: Used to create system functionality flow charts.

3 System Overview

During the initial design process (see §2), a set of requirements were derived from the system specification listed in Appendix A. The functional requirements from this set are displayed in Section 3.1 and the non-functional requirements are displayed in Section 3.2.

3.1 Functional Requirements

1. The system shall be able to allow patients and General Practitioners to manage, create and facilitate digital appointments.
2. The system shall be able to securely store sensitive patient data.
3. 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.
4. 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.
5. The system shall allow the patient records to be viewed and updated by the General Practitioner.
6. The system shall allow the General Practitioner to refer the patient to the Local Health Board if the required service is available locally, or to the National Health Board if the required system is not available locally.
7. 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.
8. The system shall allow Hospital doctors to view patient details and vitals, and add entries on diagnoses made and treatment given.
9. The system shall notify the General Practitioners when the Ambulance Service Administrator or Health Board Administrator has updated the patient details.
10. The system shall allow patients to view their vital data, and any changes made as a result of their digital appointment.

11. The system shall allow General Practitioners to order tests for a specific patient that is affiliated with the Practice.
12. The system shall allow Nurses to only view details of patients' treatments, tests or test results.
13. The system shall allow General Practice Administrators to view, make or delete appointment details between patients and General Practitioners.
14. 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.
15. The system shall allow Ambulance Service Administrators to log calls in real-time.
16. The system shall allow the Ambulance Service Administrators to dispatch an ambulance that is closest to the region where the call was logged.
17. The system shall allow the Ambulance Service Administrators to view patient records and details.
18. 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 or that the patient has been taken to A&E.
19. The system should allow Ambulance Service Crew to indicate when each call-out has been addressed.

3.2 Non-Functional Requirements

The non-functional requirements can be divided into three categories: product requirements, organisational requirements, and external requirements.

Product Requirements

1. The system shall be able to handle all users that attempt to access, update or modify patient details, treatments given or diagnoses made.
2. The system shall be able to process all identifiable data under the Data Protection Act, 1998.
3. The system shall be able to obtain real-time patient vitals in 0.1 seconds, load all operations within 1 second and process all requests within 2 seconds.
4. The system shall be available between 24 hours a day, 7 days a week, except for maintenance breaks, which should be stated at least 7 days in advance.
5. The system should have centralised logs that can maintain all services, instances and possible errors in a single location.
6. The system shall be accessible via mobile devices and personal computers.

Organisational Requirements

1. The system shall be down for maintenance, on the last working day of each calendar month, for 30 minutes.
2. The system shall be evaluated for response times on the last working day of each calendar month.
3. The system shall be able to prevent cross-scripting attacks.
4. The system will not store unencrypted sensitive data.
5. All patient data will be retained in system archives for up to 5 years.

External Requirements

1. The system should be able to support an annual growth of 10 General Practices that can utilise this platform.
2. The system should be able to support a 100% growth in user concurrency, and still meet all defined functional and non-functional requirements.

3.3 Assumptions

In order to architect the system, there were a number of assumptions that were made in relation to the scenario. These assumptions and their justifications, are listed as follows.

1. **A Health Board Administrator is responsible for redirecting a referred patient to the nearest Hospital that offers their required service.**

   A Health Board could encompass multiple hospital; for example, the Fife Health Board could include Adamson Hospital (Cupar), Victoria Hospital (Kirkcaldy), and St Andrews Community Hospital. Thus, when a patient is "referred" to a local Health Board, it is assumed that the GP does not refer the patient to a specific hospital.

   It is assumed 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 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, improving the consideration of data security within the proposed architecture.

2. **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. The architecture therefore does not consider this concern.

3. **Different members within the GP, Health Board, and Ambulance Crew have different access permissions with respects to patient data.**

   Primarily, it is assumed 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), and patient diagnosis and treatment information.

   The data which users are authorised to view is assumed to be in concordance with the access permissions listed in the scenario in Appendix A, with unreferenced data permissions being extrapolated appropriately where required.

4. **All medical personnel records are registered in the National Medical Professionals' Database.**

   It is assumed 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 assumption allows for 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.

3.4 Functionality

In order to design an architectural framework that is able to accurately execute the required functionality of the system, it was important to conceptualise the expected logical flow of events for different parts of the system.

Figures 2 and 3 display an overview of the event flows which are associated with a patient making an appointment and the system's ambulance-related capabilities respectively. These flow diagrams were used to influence the functionality which the proposed architecture was designed to support.

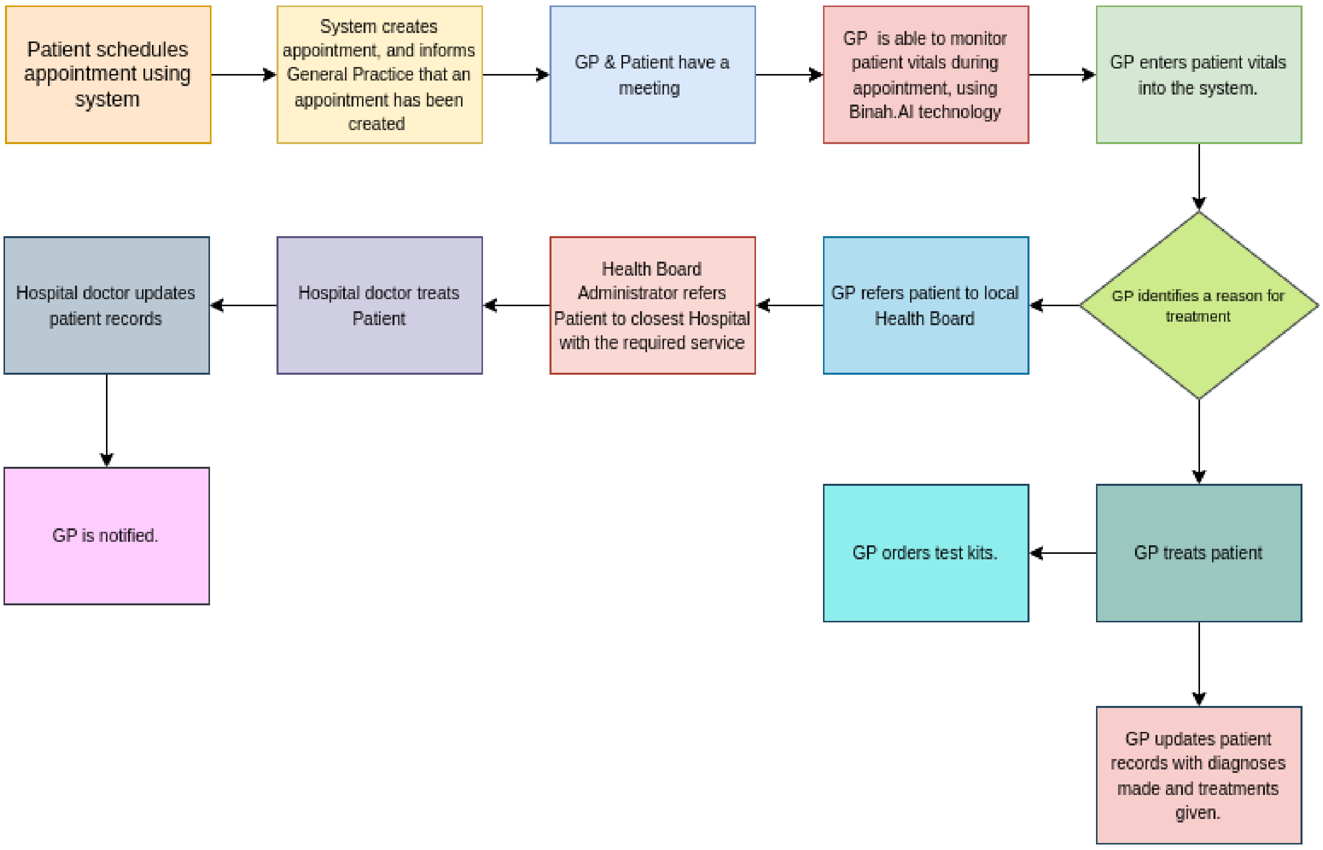
Patient schedules appointment using system
System creates appointment, and informs General Practice that an appointment has been created
GP & Patient have a meeting
GP is able to monitor patient vitals during appointment, using Binah.AI technology
GP enters patient vitals into the system.
GP identifies a reason for treatment
GP refers patient to local Health Board
Health Board Administrator refers Patient to closest Hospital with the required service
Hospital doctor treats Patient
Hospital doctor updates patient records
GP is notified.
GP treats patient
GP orders test kits.
GP updates patient records with diagnoses made and treatments given.

Figure 2: The flow of events relating how appointments are handled by the system.

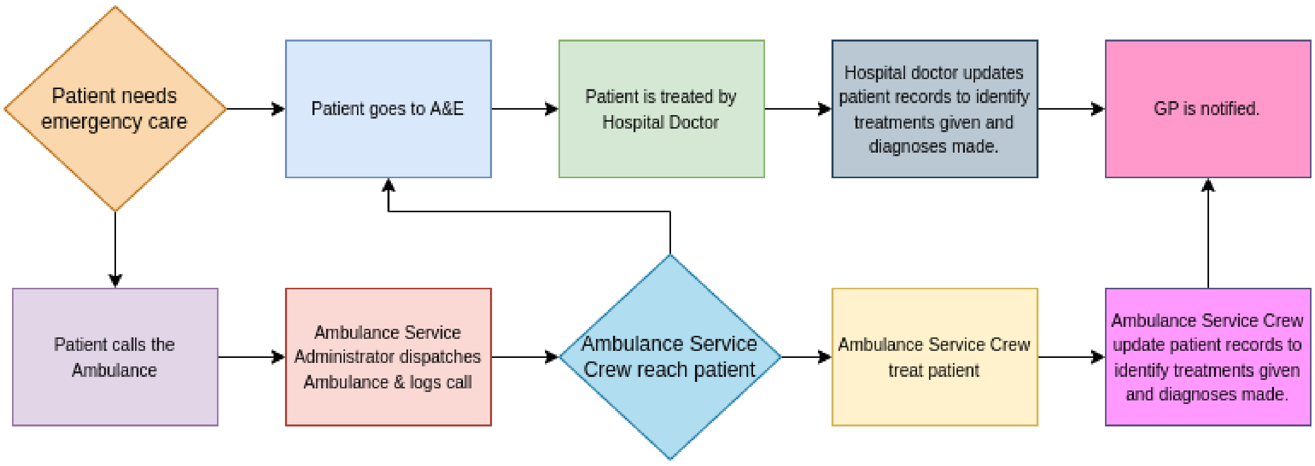
Patient needs emergency care
Patient goes to A&E
Patient is treated by Hospital Doctor
Hospital doctor updates patient records to identify treatments given and diagnoses made.
GP is notified.
Patient calls the Ambulance
Ambulance Service Administrator dispatches Ambulance & logs call
Ambulance Service Crew reach patient
Ambulance Service Crew treat patient
Ambulance Service Crew update patient records to identify treatments given and diagnoses made.

Figure 3: The flow of events relating to the system's handling of ambulance calls.

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 employed in combination with a notation similar to UML. This choice was made as all team members were relatively familiar with the concepts used in 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 [13]. The viewpoints are designed to reflect 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 4.3 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. Sections 4.4, 4.7, 4.6, and 4.5 display container diagrams for the subsystems which they describe.
- **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. Only a limited number of component diagrams are used to describe the proposed architecture for the sake of brevity; Sections 4.5 and 4.6 display a component diagram for the parts of the architecture which they describe.
- **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. No code diagrams are displayed in this report, as they are typically auto-generated from a codebase [13].

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 [13].

4.2 Notation

A UML-like notation is used within the C4 Model diagrams, as group members were relatively familiar with the technology. The notation is a hybrid between UML [14] and the suggested C4 Model notation [13] to keep the diagrams understandable and focussed. It is also worth noting that the diagrams are purely structural, they do not indicate how elements interact as this is described elsewhere only that the elements are communicating with each other.

4.3 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 provided by multiple independent, distributed services. The core services used and their roles within the architecture are as follows.

- **Appointments 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 4.4.
- **Patient Information Service**: Handles CRUD operations relating to patient information, authorising information similarly to the Appointments Service. This service is described further in Section 4.5.
- **Ambulance Service**: Affords the logging of calls and the dispatching of ambulances. This service is described further in Section 4.7.
- **GP Notification Service**: Provides the ability to asynchronously pass notifications to GPs. This service is described further in Section 4.6.

In order to enable user interaction with services, a client-server-like architectural style is employed for user-service interactions; Section 4.8 discusses this further.

Figure 4 displays the system context diagram (see §4.1), where the set of core services, users, and interactions can be seen.

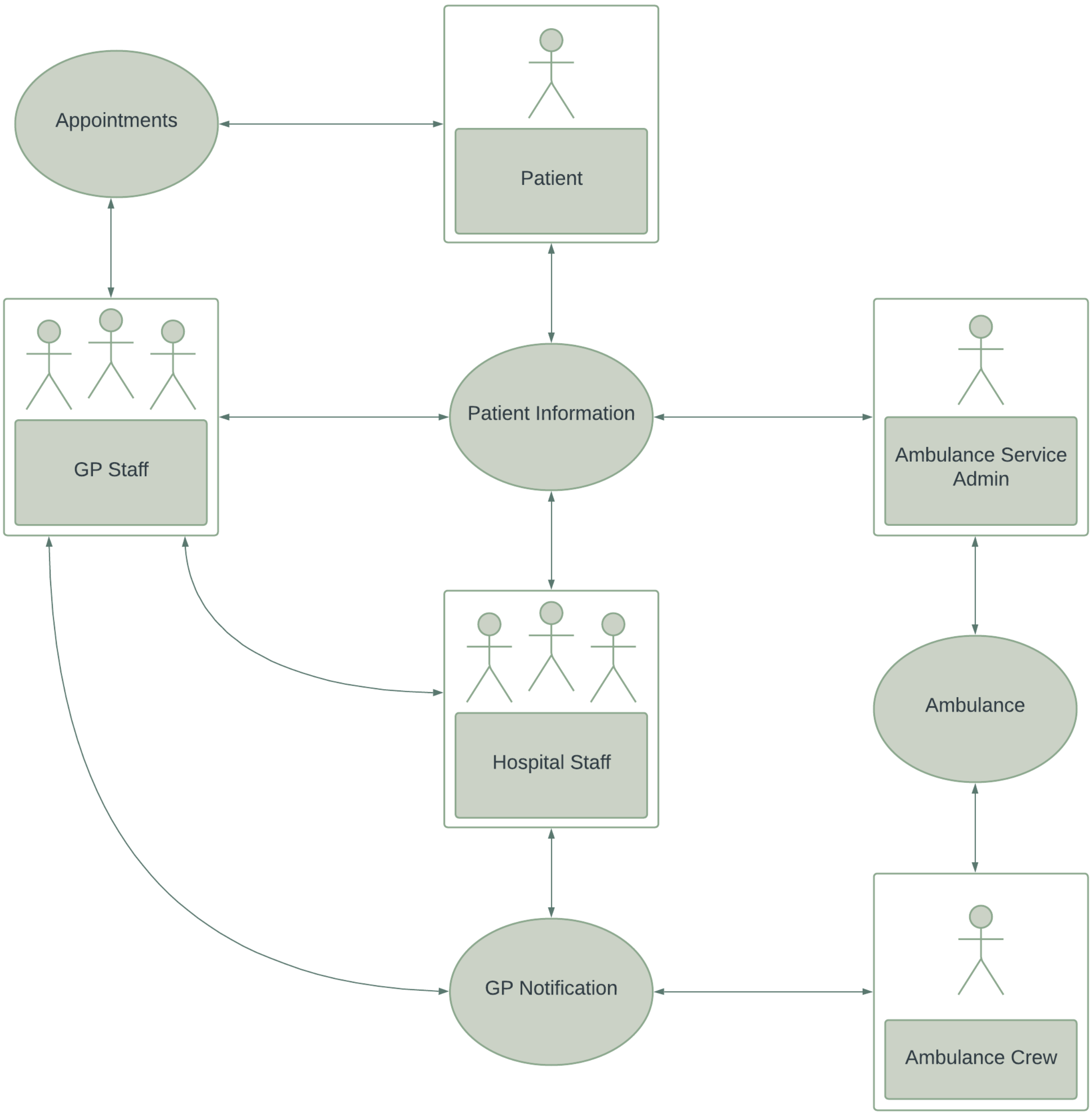

Figure 4: The system context diagram for the proposed architecture, displaying the service-oriented approach used as the fundamental architectural style for the system.

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 [15]. 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 [16]. This is primarily an advantage over more rigid architectures, such as those employing a monolithic style.
- 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 improves the system's fault-tolerance.

- 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 Hospitals (see requirement 8 in Section 3.1), 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 [16].
- 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 [15].
- 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 [15].

4.4 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 workloads, consisting of a service which contains the business logic for the Appointments Service, a datastore, and a cache. Figure 5 displays the container diagram for the Appointments Service.

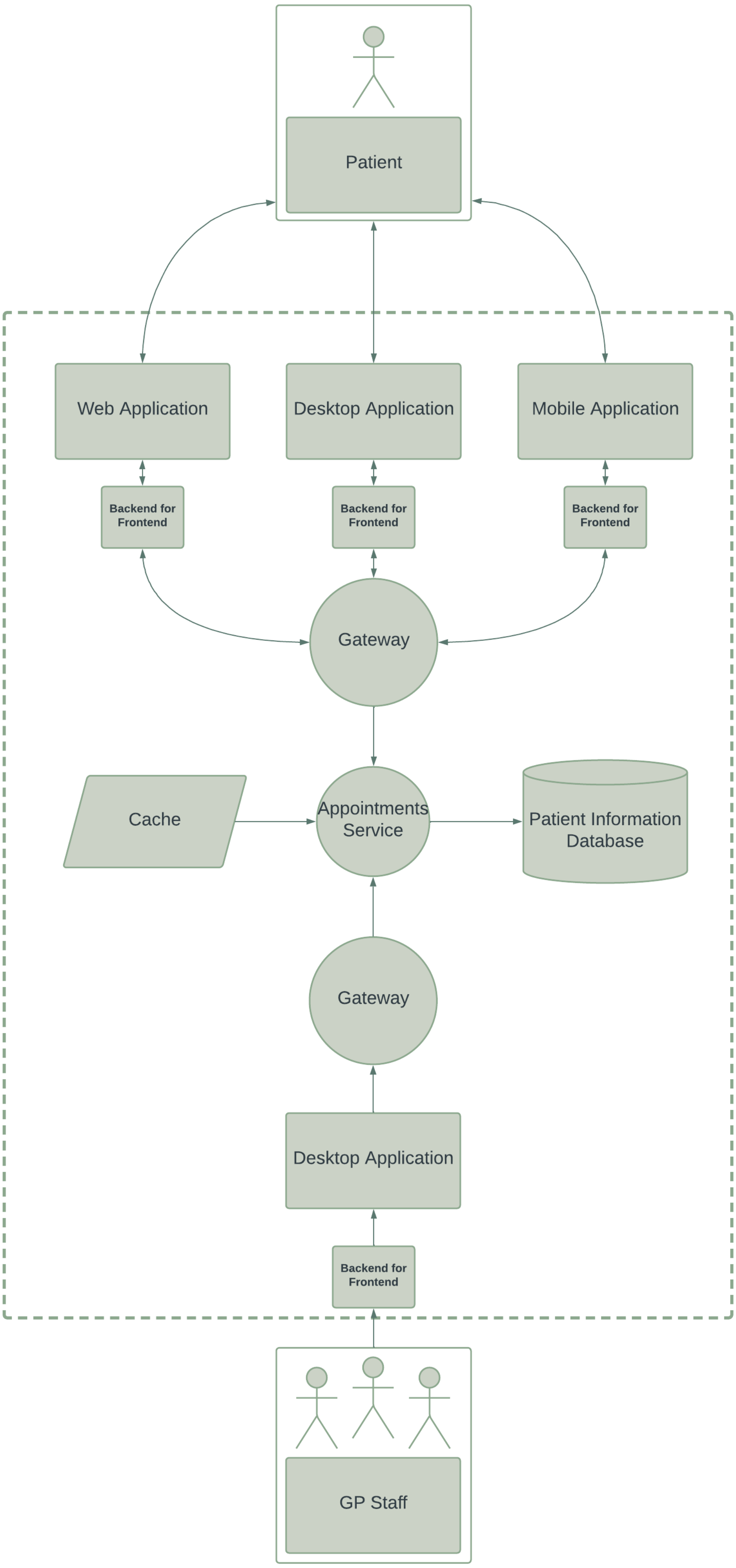

Figure 5: The Appointments Service described using a container diagram.

Datastore

The Appointments Service's datastore is used to persist data relating to appointments, improving the service's reliability. The datastore is integrated within the Appointments Service's architecture in such a way that the datastore component is 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 rules contained within the service component.

The application of the SAP also follows the advice given in Martin's Clean Code regarding how datastores should be integrated within architectures [17], meaning that the choice of data storage technology is not confined by the service's implementation, 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 after they had been retrieved from the 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 [18]. Fortunately, as the Appointments Service's workload is read-heavy, the trade-off of increased overhead is likely worth it when considering the potential performance gains introduced by the cache. Other overhead introduced by the cache, which includes the storing of entries, checking of entries, and handling evictions, is considered to be marginal when compared against the potential performance gains provided by the cache.

As it is assumed that there are many users, as the system is designed for the NHS, it is likely that evicting items from the cache on the commonly employed 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 a least-frequently used (LFU) or re-reference interval prediction (RRIP) policy, it considered as an implementation detail to determine this, and is therefore excluded from the scope of the architecture.

4.5 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 [1] and different users need different access permissions as per the system requirements (see §3.1).

Figure 6 displays a container diagram for the Patient Information Service.

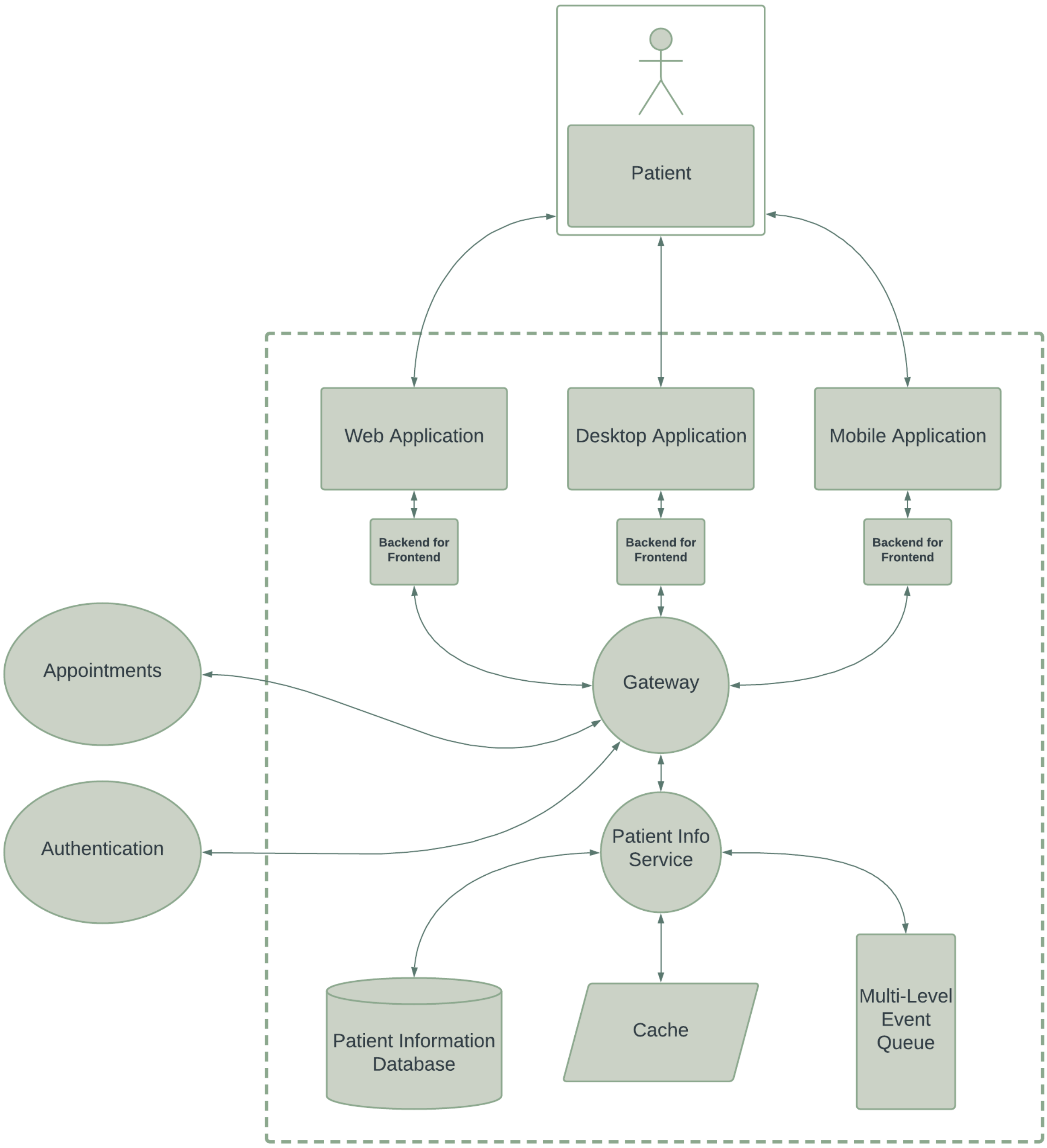

Figure 6: The Patient Information Service described using a container diagram.

One of the key components of the Patient Information Service is the Service component. Figure 7 displays a component diagram for this component, with the arrows representing the direction of dependency between the constituent parts.

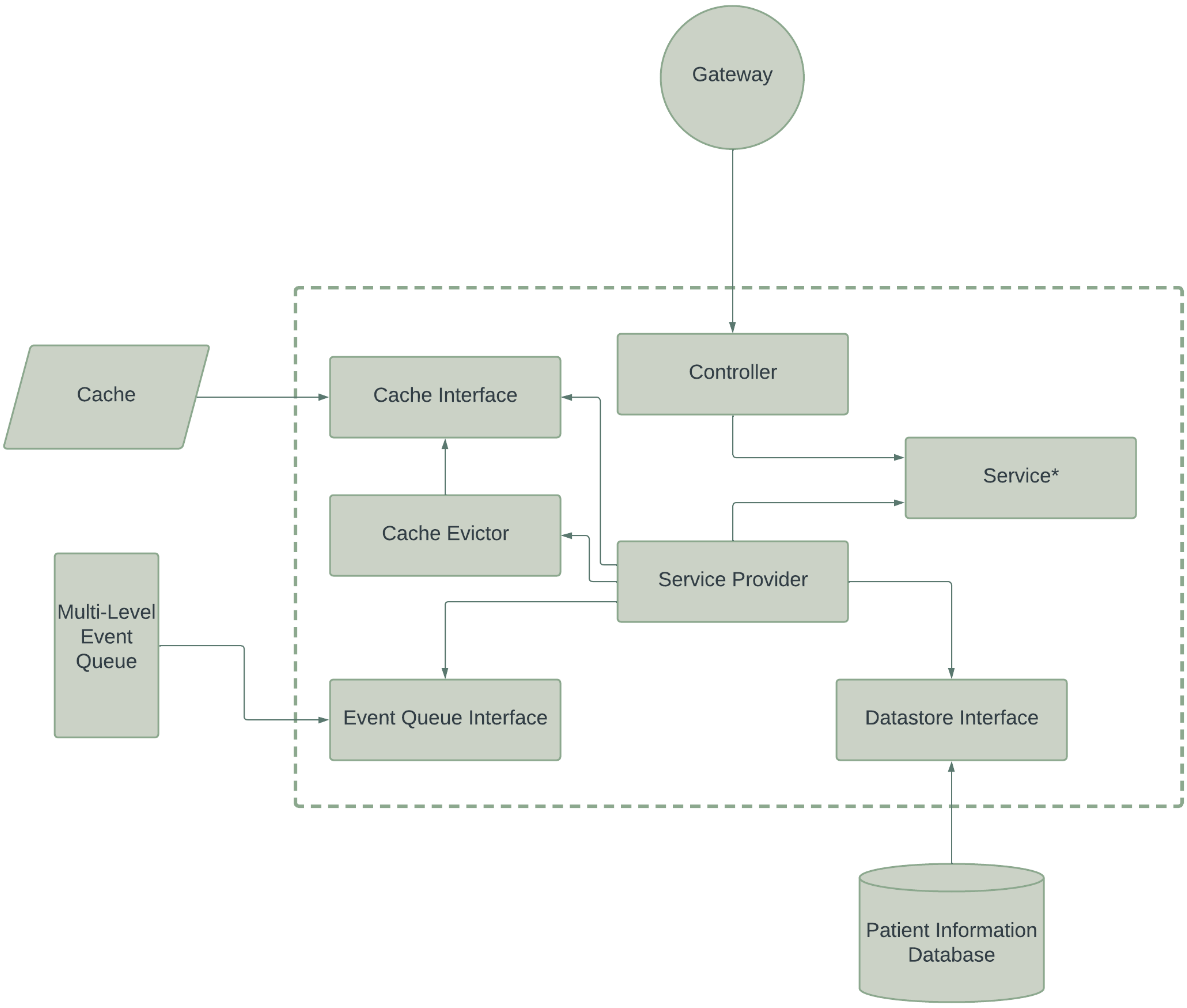

Figure 7: The Patient Information Service's Service component described using a component diagram.

It can be seen from the dependencies shown in Figure 7 that the Stable Abstractions Principle (see §2.2) is applied, such that all dependencies are pointing towards an abstract concept, such as an interface or the implementation of the business rules in the Service subcomponent (marked with an asterisk in the diagram).

Datastore & Cache

As the data handled by the Patient Information Service and the Appointments Service (see §4.4) are both related to individual patients and are associated with read-heavy workloads, the datastore and cache used within the Patient Information Service are similar to those used in the Appointments Service (see §4.4), 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 (i.e. write behaviour that experiences spikes in workload) when patient information is generated by GPs. 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.

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 proposed architecture) taking tasks from the data structure based on descending priority, meaning that more important tasks are completed before less important ones [19].

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 the updating of 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 [20], 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.

4.6 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 rest of the infrastructure which sends notifications to GPs, such as the hospital.
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 8 displays a container diagram for the GP Notification Service.

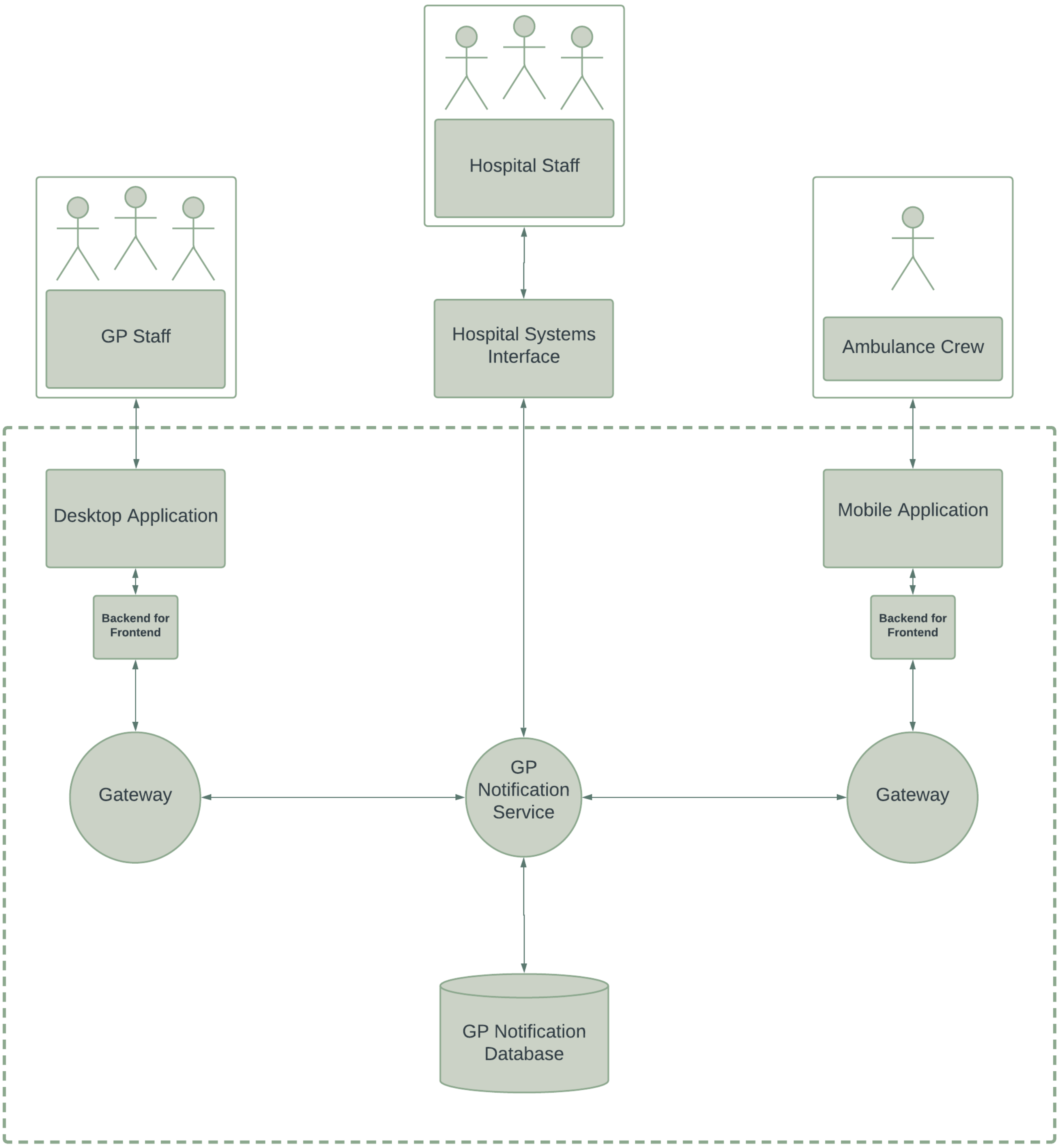

Figure 8: The GP Notification Service described using a container diagram.

One of the key components within the Appointments Service is the Service component; Figure 7 displays the component diagram for this component.

Similarly to the Service component within the Appointments Service (see §4.4), the Stable Abstractions Principle and Common Closure Principle (see §2.2) are applied, such that dependencies point towards more stable components.

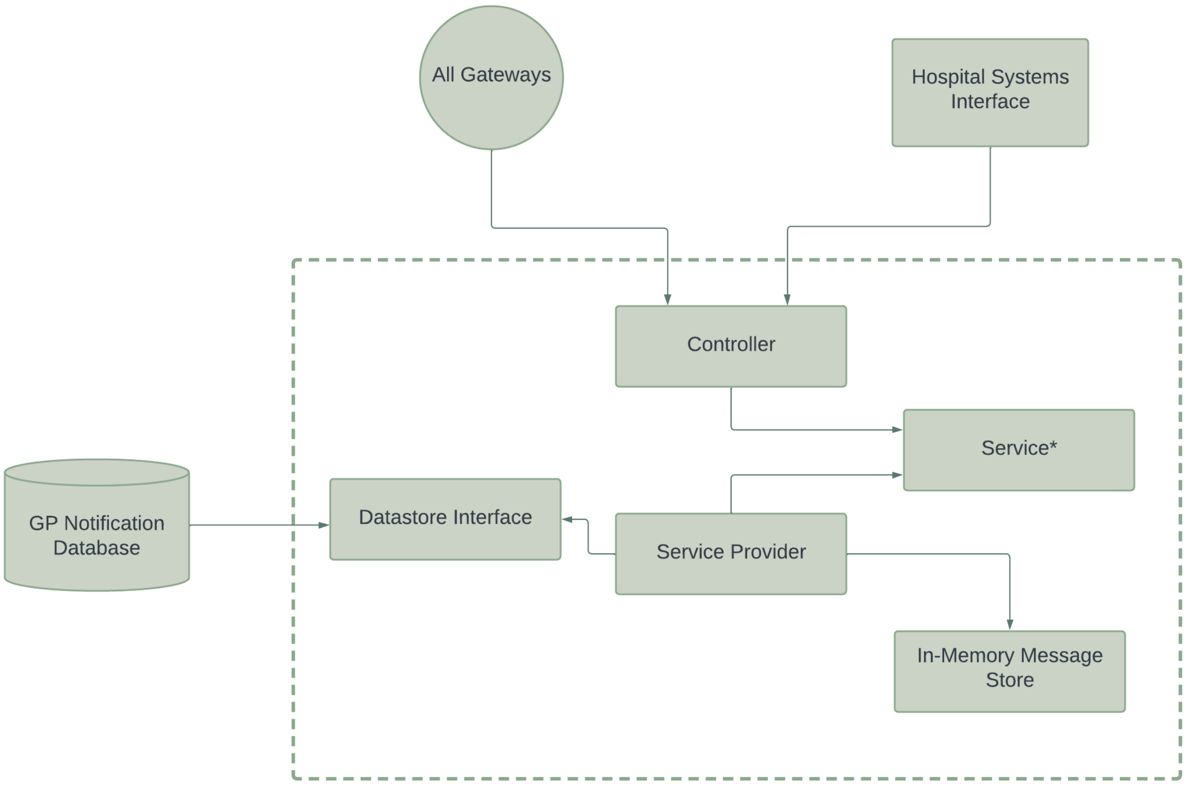

Figure 9: The GP Notifications Service's Service component described using a component diagram.

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.

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 [21].
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 given that it is a user device.
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, meaning that the delay of messages caused by a pull configuration does not detract from the functionality of the system.

4.7 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 10 displays a container diagram describing the architecture of the Ambulance Service.

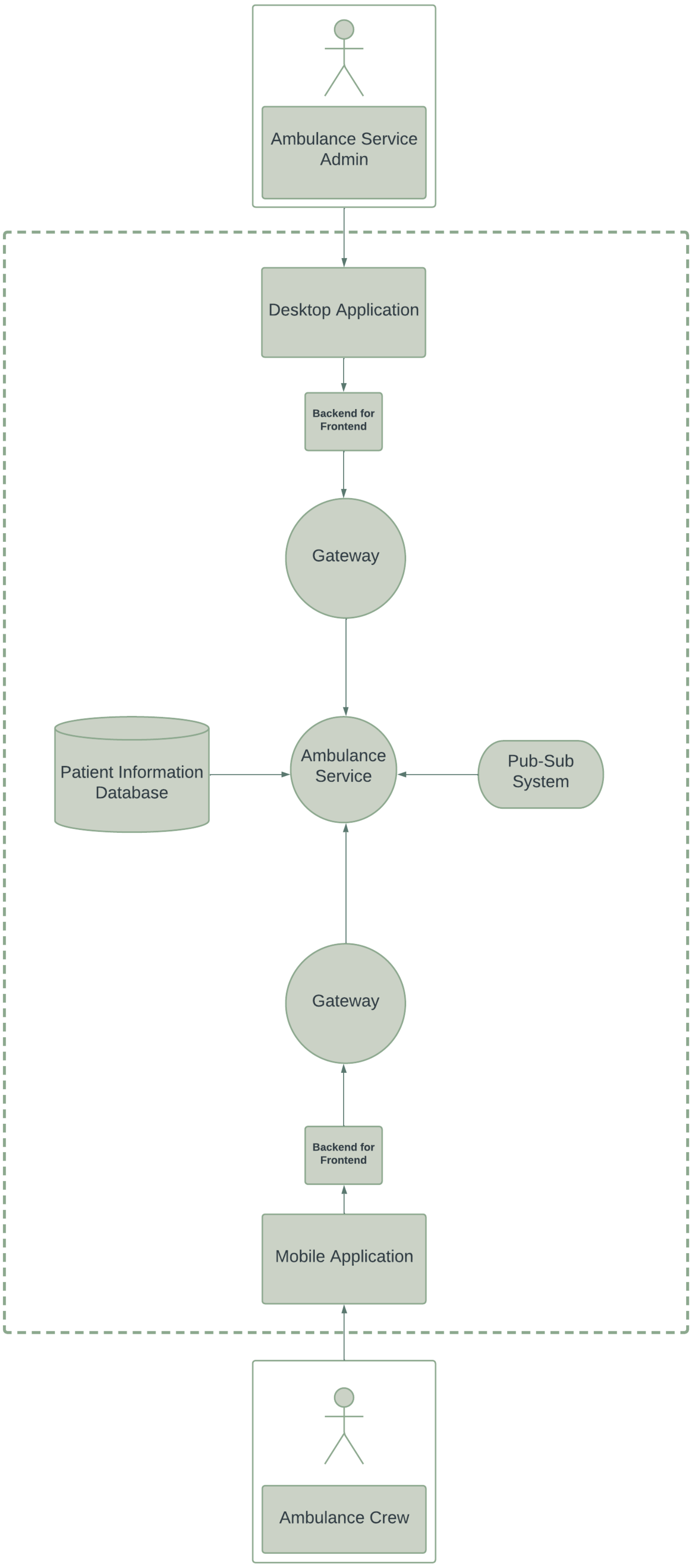

Figure 10: The Ambulance Service described using a container diagram.

Publish-Subscribe Pattern

Similarly to the GP Notification Service (see §4.6), 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 [21]. 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.

4.8 Clients

The clients within the architecture offer the functionality of the services in the form of a user interface (UI). All clients share a similar architecture, which can be seen in Figures 5, 6, 8, and 10, and provides the support for a variety of different device types, as well as the potential for functionality, such as form validation.

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.

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 [22].
- 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 [23][22].
- 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 [22].

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 [24], 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 [25].
- 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 6.1 of this report.

4.9 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, functional requirement 2 in Section 3.1 can be achieved through the use of authentication.

In order to include authentication within the system's architecture, the OAuth 2 framework [26] 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 [27]
- It enables single-sign on for the target system, which is beneficial from a user standpoint [28].
- The protocol used to authenticate users is simple in comparison to other authorisation techniques.
- It provides simple integration with third-party services [27]. 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 the specified architecture in Section 4, 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 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 afterthought. From the system specification, it can be inferred that 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 [29]. 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 [30], 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 the implementation meets all the non-functional requirements and to require a high level of security, reliability, and availability while safely managing concurrent processes, the properties must be "hoisted". George Fairbanks defines architectural hoisting as "the direct ownership, management, or guarantee by the architecture of a feature, property, or quality attribute" [31]. For example, Enterprise Java Beans (now known as Jakarta Enterprise Beans) could be employed to hoist the properties of concurrency and scalability. EJB is a Java API which has various purposes, such as managing processes and threads while running concurrency management.

Another method which will push the 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, unit tests can be used for each smallest element of the architecture. These tests will insist that each of these "building blocks" function correctly on their own before the cross-service functionality is tested. After the unit tests, integration tests can also be derived from the architecture. Each connection in the various design diagrams represents an interface between two containers, components, or classes. Integration 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, inspiration could be taken from Netflix's ChaosMonkey [32] and test the services after taking different nodes down to simulate various failures. In production, failures would be minimised due to the proposed deployment, 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 diagrams in Section 3.4, it can be seen that many interaction sequences that have two branches created at each decision point; the acceptance tests should simulate entire interaction sequences and should assert the correct output at each step.

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 [33]. When implementing the healthcare system vigilance should be taken with respects to following 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, the system could be simply rolled back 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 the 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

AWS Elastic Load Balancers (ELBs) could be utilised in the implementation of the architecture to 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 the 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 which could be utilised is the CodeDeploy system [34]. By using AWS CodeDeploy, it is possible to coordinate application deployment and updates across an arbitrary number of EC2 instances. By automating deployments, following the twelve-factor app principle of disposability, it would be possible 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 could 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 [35]. 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 the system.

Another technology that could be leveraged is React. React is a JavaScript library for front-end development that focuses on reusing components and not repeating code across classes [36]. This property aligns with the twelve-factor principle of keeping shared code within libraries that can be exposed to explicit classes which need them.

By using React for the frontend, the advantages of the React Native framework could also be exploited. React Native integrates with React, and can be used to create interfaces for multiple platforms from one codebase [37]. This would be beneficial when implementing the system across the desktop application for the GPs, the application for the hospital systems interface, and the mobile application for patients. React could be integrated with the Spring backend through the use of JSON web communication.

The backend service could be deployed as 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.

6 Evaluation

6.1 Current Considerations

Architect Biases

An important consideration when developing a software architecture style is ensuring that the architects are able to view the system context in an objective manner, in order to evaluate the expectations and requirements appropriately. However, bias can be introduced when the architects have had experience working with different architectural styles and patterns in the past, as this familiarity can sometimes cause architects to be further inclined to work with such architectural frameworks again. This makes it difficult for the architectural team to objectively evaluate the appropriate style for a particular scenario.

Due to this reason, the team had a planning phase (see S2.1), where the Architecture Trade-off Analysis Method (ATAM) was employed, which is a risk-mitigation process that is used to assess trade-offs and sensitivity points. This allowed the team to outline the desired architectural characteristics of the system, which made it much simpler to evaluate the different architectural options against these criteria, thereby enabling us to make more objective decisions. This process also enabled the team to identify the risks that needed to be considered during the development of the software, such as ensuring that the system is able to handle malicious attacks and handle data appropriately. Furthermore, this process also enabled the team to identify the different stakeholders within the system and ensure that the system was able to handle all their respective requirements.

Thus, this method was extremely useful to ensure that the team was able to view the system requirements through an objective lens, and ensure that each architectural characteristic was preserved.

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 §4.6) and the Ambulance Service (see §4.7), as well as the multilevel priority queue used in the Patient Information Service (see §4.5). 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 §4.6); 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 [21]; 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 4.9. However, as all client devices must communicate with the Authentication Service in order to access the internal services' functionality (see §4.9), 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. GP staff) 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 significant overhead to the system, meaning that replication should only be used if absolutely required [38].

BFF & Gateway Patterns

As discussed in Section 4.8, 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 [24]. 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 4.8, 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 [39].

Size

Another important point of reflection refers to the size of the teams working on each individual component of the architecture and its subsequent implication.

This is because it can often be complex to split the development process appropriately, as each team needs to be fully briefed on the architectural description, and the methodology to implement the constraints and design specified. Thus, the architectural owner or the architectural owner team must be able to effectively communicate the parameters of the architectural framework to the developers, in order to ensure that the system is developed as intended.

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 [1]. 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 §4.5). 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 §4.9).

6.2 Lifecycle Considerations

An important consideration of this system refers to its maintainability. As this system is an example of critical software, it is essential to ensure that the system is maintainable and that any additions to the system functionality are conducted in accordance with the specified architecture.

As the architecture has been designed with specific architectural characteristics in mind, it is essential to ensure that any modifications are made in accordance to the designed system architecture. This is essential to prevent architectural drift and architectural erosion, i.e., the introduction of design decisions that are not included within the prescribed architectural style/pattern, or the introduction of design decisions that are in direct violation of the prespecified architecture.

As systems evolve, and as the initial development team is succeeded by subsequent maintenance teams, it becomes much harder to maintain the sanctity of an established architectural framework, and thus, it is much easier to run into issues of architectural drift and erosion.

Thus, in order to mitigate this issue, it is essential to maintain up-to-date documentation, that is appropriately updated, whenever changes are made or new functionality is added. As such, it should be essential to update the documentation whenever any changes to the system are made.

Furthermore, the application of component principles in the development of this system architecture is a simple way to address this issue, as developers can ensure that they are familiar with the architectural requirements of each component before they modify or update it.

As different aspects of functionality are contained within a modular structure, it is extremely easy to update these capabilities in an insular manner, and ensure that any changes work with the rest of the system, before release. This can be done through Continuous Integration/Continuous Development (CI/CD) pipelines. Furthermore, the component architecture of this system enables functionality to be reused throughout the system, with the assurance that this functionality has been tested in a robust environment, which is an essential criterion for a critical system.

7 Conclusion

This report provides an overview of an architecture for the system described in Appendix A, along with a detailed breakdown of architecting process and system requirements, an implementation plan, and a critical evaluation of the proposed architecture.

The architecting process described in Section 2 is detailed, and the requirements analysis completed in Section 3 is thorough, and considers situations which could be considered ambiguous in the original specification.

The proposed architecture itself employs a wide variety of both common and unusual architectural features, patterns, and styles to shape the proposed architecture towards the needs of the target system, including the application of the client-server pattern (see §4.8) and a multilevel priority queue (see §4.5).

The evaluation completed for the architecture and the proposed implementation plan for the architecture in Section 5 are both detailed and demonstrate substantial knowledge of the software architecture paradigm.

Overall, this practical has been completed to a high standard and fulfils the base specification, as well as completing several minor extensions.

References

[1] "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

[2] "Cyber and data security, NHS Digital," Accessed: 2022-03-27. [Online]. Available: https://digital.nhs.uk/cyber

[3] "The HIPAA Privacy Rule, HHS.gov," Accessed: 2022-03-29. [Online]. Available: https://www.hhs.gov/hipaa/for-professionals/privacy/index.html

[4] I. o. M. U. S.), S. J. Nass, L. A. Levit, L. O. Gostin, S. J. Nass, L. A. Levit, and L. O. Gostin, *Beyond the HIPAA privacy rule: Enhancing privacy, improving health through research.* National Academies Press, 2009.

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

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

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

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

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

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

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

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

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

[14] "About the Unified Modeling Language Specification Version 2.5.1, Object Management Group," Accessed: 2022-03-23. [Online]. Available: https://www.omg.org/spec/UML/

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

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

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

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

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

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

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

[22] "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

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

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

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

[26] "OAuth 2.0," Accessed: 2022-03-27. [Online]. Available: https://oauth.net/2/

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

[28] "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

[29] E. Whiting and S. Andrews, "Drift and erosion in software architecture: Summary and prevention strategies," in *Proceedings of the 2020 the 4th International Conference on Information System and Data Mining*, ser. ICISDM 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 132–138. [Online]. Available: https://doi.org/10.1145/3404663.3404665

[30] "Principles behind the Agile Manifesto," Accessed: 2022-03-27. [Online]. Available: https://agilemanifesto.org/principles.html

[31] "Architectural hoisting (original blog post), George Fairbanks," Accessed: 2022-03-27. [Online]. Available: https://www.georgefairbanks.com/blog/architectural-hoisting-original/

[32] "Chaos Monkey," Accessed: 2022-03-29. [Online]. Available: https://netflix.github.io/chaosmonkey/

[33] "The Twelve-Factor App," Accessed: 2022-03-27. [Online]. Available: https://12factor.net

[34] "AWS CodeDeploy," Accessed: 2022-03-29. [Online]. Available: https://aws.amazon.com/codedeploy/

[35] "Spring," Accessed: 2022-03-29. [Online]. Available: https://spring.io/

[36] "React - A JavaScript library for building user interfaces," Accessed: 2022-03-29. [Online]. Available: https://reactjs.org/

[37] "React Native - Learn once, write anywhere," Accessed: 2022-03-29. [Online]. Available: https://reactnative.dev/

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

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

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.