Patterns for Software Systems in Low-Resource Environments

ABAYOMI O. AGBEYANGI, University of Cape Town, Chrisland University
HUSSEIN SULEMAN, University of Cape Town

This paper presents a collection of patterns designed to address challenges in software systems development for low-resource environments, particularly prevalent in low-income countries. These environments are characterised by constraints such as low-end devices, limited data accessibility, power instability, low literacy, a lack of expertise, and language barriers. The patterns offer practical solutions to optimise software efficiency while managing scarce resources. By incorporating lightweight development to make more efficient use of available resources, an audio-visual interface for better interaction for underserved users and language localisation to improve accessibility, user engagement, and cultural sensitivity, software systems can operate optimally under resource limitations. Case studies illustrate the successful implementation of the patterns, demonstrating their capacity to deliver critical services and improve user experiences. The patterns provide valuable insights to software development, empowering software developers to build more efficient and resilient solutions for low-resource environments.

Categories and Subject Descriptors: D.2.2 [**Design Tools and Techniques**]: Software design techniquess—*Software design engineering*; D.2.13 [**Reusable Software**]: Reuseability—*Reuseability*; D.2.11 [**Software Architectures**]: Software architecture—*Software Design*

General Terms: Low-resource environment

Additional Key Words and Phrases: low-resource environment, software design, software systems, software challenges, patterns

ACM Reference Format:

Patterns for Software Systems in Low-Resource Environments. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2023), 11 pages.

1. INTRODUCTION

Software systems are extremely important in numerous areas of our lives, including medical care, education, transportation, entertainment, etc. Despite this, developing software systems that are capable of operating under conditions with limited resources in an efficient and effective manner can be an overwhelming task [Meyer et al. 2020].

In low-resource environments, such as disconnected regions, low- and middle-income countries, or regions that have suffered from natural disasters, dependable power sources, access to essential data, the availability of skilled experts, and the ability to understand the software in the local language due to a low literacy level are typical issues [Di-Pietro et al. 2020]. Software systems that can function effectively and efficiently in such circumstances must be adequately planned for.

In this paper, we describe patterns for developing software systems for low-resource environments when challenges like low-end devices, a lack of adequate power supply, the unavailability of skilled experts, inadequate

This work is supported by National Research Foundation of South Africa (Grant numbers: 105862, 119121 and 129253) and University of Cape Town.

Author's address: A.O. Agbeyangi, Department of Computer Science, University of Cape Town, Rondebosch, Cape Town, South Africa and Chrisland University, Abeokuta, Nigeria ; email: abayomi.agbeyangi@uct.ac.za; H. Suleman, Department of Computer Science, University of Cape Town, Rondebosch, Cape Town, South Africa; email: hussein@cs.uct.ac.za

data accessibility, and language barriers are pertinent. These patterns were derived from experiences developing software systems in low-resource environments ([Suleman 2021; Suleman 2022]), generalising the concept and approach as a framework for software systems in low-resource environments.

The aim is to address the challenges encountered in developing software systems for low-resource environments by presenting a usable and tested set of patterns. These patterns are derived from experiences within low-resource environments, forming a generalised framework to guide software development in such contexts.

The significance of developing software systems for low-resource environments lies in their potential to empower communities with critical software needs to ease existing complicated manual processes, even amidst challenging conditions. By addressing issues like low-end devices, inadequate power supply, lack of expertise, limited data accessibility, and language barriers, these software systems can have a transformative impact on socio-economic development and significantly improve lives within the regions.

The target audience for these patterns is software developers and institutions or organisations with an interest in developing software systems for low-resource environments, particularly those within the region.

The following sections start with detailed descriptions of the challenges, describe the patterns and use cases, and conclude with a discussion of the pattern application and future work.

2. METHOD

The patterns were designed based on the challenges associated with developing and using software systems in low-resource environments. The pattern structure is in a design pattern format, with each pattern consisting of a name, a description of the problem, a solution, and a set of related patterns.

2.1 The Challenges

Low-resource environments are characterised by insufficient infrastructure, scarce resources, and diverse user needs. The absence of dependable and robust infrastructure can have significant negative effects on software systems [Bon et al. 2016]. The power supply might be erratic [Abou-Khalil et al. 2021; Di Pietro et al. 2020], internet connectivity could be slow or intermittent [Ramanujapuram and Malemarpuram 2020; Sengupta et al. 2021], and hardware resources may be limited [Gautam et al. 2017], resulting in outdated or restricted devices. The linguistic and cultural diversity often leads to populations with varying levels of literacy and digital skills [Kowalski et al. 2022]. Sustaining and supporting software systems in low-resource environments thus comes with other challenges such as inadequate technical expertise, insufficient funds for training or hiring skilled personnel, changing user needs, evolving technology, and challenges in accessing regular updates or bug fixes.

For instance, in a small health clinic in a remote, low-resource environment that relies on an eHealth software system to manage patient records, appointments, and inventory. The clinic experiences frequent power outages due to an unreliable electrical grid. During one such outage, the clinic‘s power supply was disrupted for several hours. The power outage directly impacts the software system by causing system downtime, data loss or corruption, system instability, and decreased reliability. Thus, clinic staff may resort to manual or alternative methods to perform tasks that are typically automated by the software system. In the same vein, the unreliable internet connectivity, another prominent challenge, was emphasized by [Ofori-Manteaw et al. 2022], who stated that the ability to connect to the internet, a necessary condition for online teaching and learning, is still a problem in low-resource environments like Ghana, which is a result of the poor infrastructure, stating that only 48% of Ghana‘s population had access to the internet as of 2020.

As a result of the challenges mentioned, we further understand that each of the challenges can also have a positive or negative impact on others. For instance, insufficient capital can lead to an inadequate power supply, and vice versa. Low literacy can lead to a lack of skilled expertise, which can further lead to data accessibility issues. Hence, this can result in a chain of effects and reactions, as shown in Figure 1.

It is evident that low-resource environments present a unique set of challenges for software systems. These challenges require a comprehensive understanding of the environment and the needs of the users. Designing

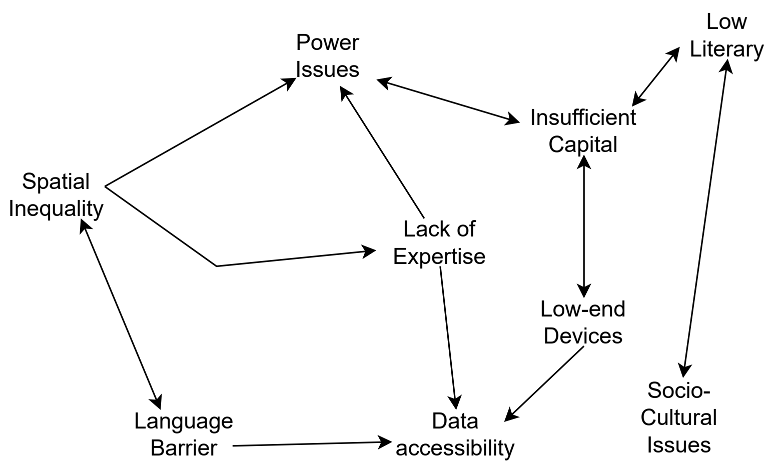

Fig. 1. Categorisation and interaction of the challenges

and implementing software systems for low-resource environments necessitates creativity, innovation, and a deep understanding of the technologies and resources available. Furthermore, effective support and maintenance of software systems in low-resource environments require a proactive approach that addresses potential issues before they arise. Proper planning, resource allocation, and effective communication with users are critical components of successful software systems in low-resource environments. It is crucial to recognise the complexity of low-resource environments and the impact they can have on software systems and to develop strategies that can overcome these challenges and ensure that the software systems continue to operate successfully.

We looked at the various issues and their chain of reactions, and the following issues were specifically chosen for consideration:

—Low-end devices

—Low literacy

—Lack of expertise

—Language barrier

—Data accessibility

2.2 Developing challenges to patterns

The patterns are designed as a solution to problems in software systems in low-resource environments arising from the chosen challenges. Three patterns describe the solutions to the challenges, as illustrated in Figure 2.

—**Lightweight Development** is a solution for developing software systems with limited resource utilisation features, in cases of low-end user devices, lack of expertise and lack of data accessibility. This can be in the form of *lightweight applications*, *lightweight operating systems* and *lightweight data* [van Tonder et al. 2008; Chavarriaga et al. 2017; Idani 2022; Scholz et al. 2023].

—**Audio-Visual Interface** is a solution when dealing with the issue of low literacy [Metatla et al. 2019; Thinyane and Bhat 2019; Shrivastava and Joshi 2019; Rayed et al. 2023].

—**Interface Localization** addresses the issue of language barriers, thus making software more useful by adapting to the users' preferred local language [von Holy et al. 2017; Moodley 2020; Telemala and Suleman 2021; Rocha et al. 2022].

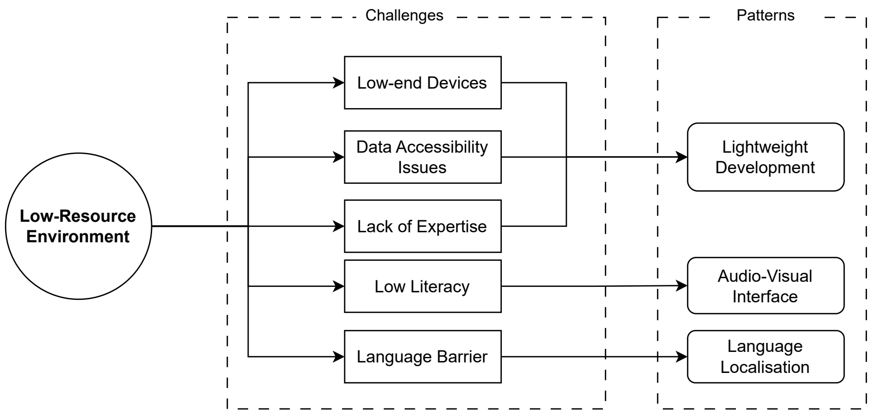

Fig. 2. Map of Patterns

Lightweight development is an approach that centres on creating resource-efficient software systems so as to address the distinctive challenges that low-resource environments present. This approach entails optimising multiple facets of software development, such as code, dependencies, technologies, and data management [Jwo et al. 2021; Sahlabadi et al. 2022]. The objective of lightweight development is to reduce resource consumption while simultaneously maximising performance, with the ultimate goal of rendering software systems more appropriate for deployment in low-resource environments. In low-resource environments, software systems' selection of lightweight technologies is of paramount importance. The judicious selection of technologies, frameworks, and platforms with lower resource requirements can guarantee optimal software system performance. This, we consider, entails the adoption of *lightweight applications*, *lightweight operating systems*, and *lightweight data management*. It emphasises efficiency, minimalism, and reducing the overall system footprint [Suleman 2019]. For example, in the use of mobile devices (low-end device), which have resource limits in comparison to desktop systems, which are typically used as an alternate computing device in low-resource environments. Mobile applications can be built to use less memory and CPU, as well as optimise network connection, by using lightweight development practises.

Audio-visual interfaces can be highly beneficial for software systems in low-resource environments. These interfaces help overcome language and literacy barriers, making software more accessible to users with limited education or literacy [Haji 2017; Thinyane and Bhat 2019; Rayed et al. 2023]. They provide intuitive interaction through visual elements and audio prompts, allowing users to navigate and perform actions more naturally. More so, audio-visual interfaces are optimised for low bandwidth and hardware limitations, making them suitable for environments with limited network connectivity and processing power. They also support multilingual interactions by providing prompts and cues in different languages, enhancing usability and inclusivity. These interfaces can serve instructional and training purposes by using audio instructions and visual demonstrations to help users understand and learn, particularly in regions with limited formal training opportunities.

The audio-visual interface pattern enables users to engage with and receive information from the software system in an efficient and straightforward manner by leveraging auditory and visual information. It is composed of audio interaction, visual interaction, and audio-visual synchronisation. In-car infotainment systems, for example, frequently use a combination of audio and minimalistic visual interfaces to present drivers with pertinent information such as navigation instructions, audio playback controls, and vehicle status indicators. This works well for both literate and nonliterate drivers. By leveraging audio and visual interfaces, audio-visual interface pattern can enhance accessibility, improve user experience, and accommodate various user needs and preferences, particularly for low-literacy users.

Interface localization, which involves adapting a software interface to the language, culture, and preferences of the target audience, is indeed useful for software systems in low-resource environments [von Holy et al. 2017].

Firstly, it increases accessibility by enabling users with limited proficiency in the software's default language to understand and navigate systems more easily. This reduces barriers to adoption and promotes usage. Secondly, localised interfaces enhance the user experience by presenting the software in the user's native language, making it more intuitive and user-friendly. It improves usability, aligns with cultural norms, and fosters higher user satisfaction and engagement. For example, take a non-profit organisation developing a mobile application to provide healthcare information and resources to rural communities in a low-resource environment where access to healthcare services is limited. The organisation aims to localise the application to multiple languages spoken in the target regions to ensure effective communication and accessibility for the local population. The development team designed the application's codebase to be easily adaptable for localization. The organisation collaborates with local translators who are fluent in the languages spoken in the target regions. The development team creates resource bundles for each language, storing the translated content. These bundles are lightweight and optimised to minimise memory usage. The application provides a language selection option on the initial setup screen, allowing users to choose their preferred language. The language selection feature is simple and intuitive, requiring minimal computational resources. Given the low-resource environment, the development team optimises the localization implementation to minimise the application's memory and storage requirements. Thus, it enables the mobile application to effectively serve the target communities where users can access healthcare information and resources in their preferred languages, leading to better understanding, engagement, and improved healthcare outcomes.

In particular, based on Figure 2, to be able to apply the patterns in a situation where there are low-end devices, data accessibility issues, a lack of expertise, low literacy, and language barriers to support software system development, the critical starting point would be to first define the requirements and constraints by understanding the specific requirements and constraints of the low-resource environment. The next step is prioritising core functionality to eliminate unnecessary features or components that may increase resource consumption. This should be followed by the necessary optimisation (on both the client and server sides) through a responsive design approach. Continuous monitoring and optimisation are also essential to monitoring the application's resource usage, performance, and user experience. More details on using each pattern is addressed in patterns section.

3. PATTERNS

3.1 Lightweight Development

Software systems must be designed to make the best use of available resources. Lightweight systems are advantageous since they consume fewer resources than other variants. A lightweight text editor, for example, may consume less memory and computing resources than a more comprehensive word processor. In comparison to their more feature-rich software systems, lightweight applications often need less processing power, memory, and storage. For instance, because it can operate faster and use fewer resources than a more feature-rich browser, a lightweight web browser might be a useful option for a device with limited processing power and memory.

How can software systems in low-resource environments be developed using a lightweight development approach to use limited resources effectively?

The issue of scalability becomes a problem in the absence of lightweight development. As the system grows, a software developer may need to upgrade software that relies on resource-intensive packages. It will necessitate more resources and, if these resources are not available, the system may become unmanageable. This happens often in low-resource environments where the software will cease to function maximally and later be abandoned as it might make adding new features or capabilities, as well as supporting more users, challenging.

More so, operating systems that are not designed for environments with limited resources might cause performance problems. Particularly for low-end devices, this might reduce the effectiveness of the software applications running on them and hurt the overall user experience.

Therefore,

To make software systems make more efficient use of available resources, leading to better performance, greater stability, and more sustainable resource usage, implement software systems as lightweight applications within lightweight environments.

For web applications, a lightweight development framework, such as Flask[1] or Express.js[2] can be chosen. Consider the user environment to identify the important features and functions necessary for the software system. The software is then designed and developed, with an emphasis on code optimisation approaches such as reducing superfluous calculations and optimising algorithms. This can be accomplished by minimising dependencies and utilising lightweight libraries to reduce the application's overall resource consumption. Test the programme on low-powered devices like mobile phones and optimise its performance and resource utilisation depending on the results (Figure 3).

As a specific example, Simple DL [Suleman 2021] was developed as a toolkit for creating simple digital libraries. It emphasises sustainability and the ability to recover from disaster as critical requirements for digital library software systems, particularly for archivists working in low-resource areas who require systems that perform without a lot of processing power or maintenance. Its metadata is saved in spreadsheets and XML files, which eliminates the requirement for a database management system. Flat files are used to store unstructured data, and XML is used to store structured data. Thus the system is designed to use lightweight data, processed with lightweight applications and aimed at lightweight hosting environments.

While it is vital to note that a database may be required for an application, one should consider lightweight data by first assessing the data storage available at the end-user system and the software system's data requirements. Lightweight data management tools such as Redis[3] or SQLite[4] can be used to efficiently store and access data while consuming few resources. It is also important to use offline storage techniques to store frequently accessed data locally, minimising reliance on network connectivity and enhancing responsiveness. To ensure optimal resource consumption, you must additionally monitor and optimise data storage and retrieval procedures.

As an example, suppose a developer wants to create an agriculture information system that can run efficiently on low-power devices used by farmers in rural areas. This could be a web-based farm information system using a lightweight application development framework such as Express.js. Services like crop guidance, weather updates, and market prices can be included. The code can be optimized by employing efficient techniques and avoiding superfluous computations. To reduce the programme's size and resource requirements, use lightweight libraries and modules. Finally, test the application on several low-powered devices to verify smooth performance and optimal resource utilisation.

Developers can design lightweight software systems that satisfy the specific demands of users in low-resource environments in an efficient and effective manner by first analysing the requirements and then taking a lightweight development approach. Plan the application architecture and create the essential components with the tools chosen. Finally, deploy the application and monitor performance, making any necessary improvements after testing and optimising it for optimal performance and resource consumption.

When compared to a standard heavyweight framework, the lightweight development pattern reduces the average time required to build and deploy new features by about 30% [Suleman 2021; Suleman 2022]. The code is much smaller, resulting in a reduction in memory usage, while benchmarks for system performance demonstrate an improvement in reaction times and a reduction in resource utilisation.

[1]https://www.fullstackpython.com/flask.html
[2]https://expressjs.com/
[3]https://redis.io/
[4]https://www.sqlite.org/index.html

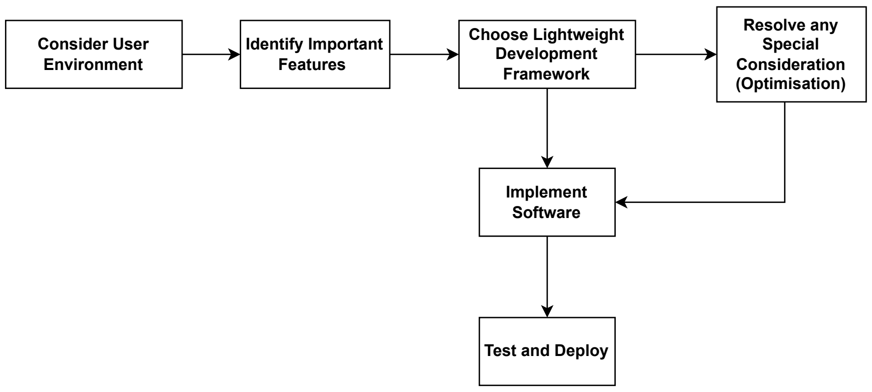
Consider User Environment
Identify Important Features
Choose Lightweight Development Framework
Resolve any Special Consideration (Optimisation)
Implement Software
Test and Deploy

Fig. 3. Lightweight Development Process

Another approach and pattern is the microservices architecture [Brown and Woolf 2016; Waseem et al. 2020], a pattern that structures an application as a collection of loosely coupled services, promoting scalability, modularity, and lightweight code components.

3.2 Interface Localization

In today's globalised society, people from many origins and cultures use software systems. Software must be designed to be accessible and convenient for all users, regardless of language or cultural identity. Interface localization can be an important technique for improving the accessibility, usability, and interaction of software systems in low-resource environments.

Interface localization is essentially intended to improve the accessibility of software systems by allowing users to communicate with the software in their native language. This is especially important for users with low literacy levels who may not be fluent in languages such as English. Software can be made more user-friendly and understandable by including regional cultural references and tailoring user interfaces to local preferences. This could lead to increased adoption and utilisation, particularly in low-resource contexts.

How can software systems in low-resource environments use interface localization to increase user engagement by creating a more personalised experience?

In low-resource environments where software systems do not use interface localization, it may have a negative effect on the effectiveness and usefulness of the software system as well as the users' engagement. Language barriers can lead to misunderstandings, errors, and misinterpretations, which can have devastating effects in some cases.

For example, a misconception caused by the language barrier of healthcare workers in local healthcare facilities using an eHealth application may result in a wrong diagnosis, inadequate care, or even fatalities. Furthermore, software systems that do not use language localization may not be sensitive to cultural differences or appropriate for specific user groups. This could result in a violation, misunderstanding, or exclusion, all of which would harm the software's effectiveness and reputation.

Therefore,

To improve accessibility, user engagement, and cultural sensitivity, software systems in low-resource environments should include interface localization.

The developer must first determine the text and other information that must be localised in the software system that is being developed. Most of this will be language in the user interface. After determining the content that can be translated, it can be extracted and transformed into a format that can be automatically translated using machine translation technologies. Once completed, the translations can be assessed in order to improve their effectiveness as well as the quality of future translations. Language localization is said to be an iterative process that may necessitate multiple iterations to ensure that the software system is appropriately translated and optimised for the target language(s). Interface localization can be achieved with the assistance of tools such as GNU gettext, the Google Translate API, etc.

GNU gettext[5] is a popular localization toolkit that offers tools for collecting and handling translatable texts from software systems. It supports numerous file formats, including C, C++, Python, and Java programming language files, and includes tools for creating message catalogues, merging translations, and developing multilingual applications. Another option is the Google Translate API[6], which uses a machine translation service that may be linked with software systems to deliver speedy and automated translations of language texts. It supports over 100 languages, some of which are low-resource languages, and provides a variety of translation models. The API is accessible via a REST interface and supports a variety of programming languages.

3.3 Audio-Visual Interface

The implementation of an audio-visual interface is extremely valuable for software systems in low-resource environments, particularly in rural areas. It addresses the problem of limited access to traditional input devices such as keyboards and mice by providing an alternate mode of engagement. Users with low literacy or physical challenges can effectively interact with software using voice commands and visual images. Audio-visual interfaces can also overcome language barriers by adding pictures, symbols, and spoken instructions, allowing users who may not be fluent in the software's language to navigate and comprehend the system. As a result, audio-visual interfaces are an inclusive solution that caters to a broad user base in poor and rural regions with limited resources.

In a region with high illiteracy rates, such as those found in most rural areas of low-resource environments, text-based interfaces can present significant challenges. An audio-visual interface mitigates this issue by minimising the reliance on written text and emphasising audio and visual elements. Users can engage with the software via spoken commands and receive responses via visual alerts, allowing people with weak reading skills to use systems effectively.

How can software systems developed for low-resource environments take advantage of audio-visual interfaces for better interaction with underserved users?

Without an audio-visual interface, software systems that rely mainly on written text may exclude those who are illiterate. Illiterate users may find it difficult to navigate and understand the system, limiting their ability to access information, complete tasks, and take advantage of the software systems's features. This often compounds the digital divide while also limiting prospects for education, skills development, and empowerment in low-resource environments.

Therefore,
Enable software systems developed in low-resource environments, particularly in underserved regions, with an audio-visual interface.

Elderly people living in low-resource environments frequently do not have caregivers and are increasingly not part of extended families. For example, many elderly people in South Africa live alone in rural areas while the younger generation moves to the cities for work. In such cases, audio-visual interfaces can enable interaction with mobile devices for communication with family and especially to seek help in emergency situations. Such applications can be expanded to include more functionality to support the elderly in home environments in general. A typical example is using speech recognition libraries and appropriate user interface design tools such as CMU

[5]https://www.gnu.org/software/gettext/
[6]https://developers.google.com/ml-kit/language/translation

Sphinx[7], Google Cloud Speech-to-Text[8], or Microsoft Azure Speech Services[9], and UI design tools like Adobe XD[10], Figma[11], or Sketch[12].

Haji Ali Haji designed a mobile graphic-based reminder system to support compliance with tuberculosis treatment regimes for patients with low literacy levels, using Zanzibar as the case study [Haji 2017]. This was implemented on the Android platform and works effectively to communicate the need to follow their treatment instructions. It was discovered that patients in the mobile device graphic-based group adhered to their treatment more than those in the traditional telephonic reminder group. Such a design intervention can have enormous benefits for healthcare in rural areas.

4. DISCUSSION

In low-resource environments, such as those in low-and middle-income countries, there are challenges in accessing educational, healthcare and other essential software systems. In these environments, software systems must be designed to operate efficiently on low-end devices, minimize energy usage, and reduce the need for data storage and network access. To address these challenges, software designers can incorporate lightweight development, audio-visual interfaces and interface localisation. A lightweight application enables software systems to run efficiently on old or low-powered computers, requiring minimal system resources. In a low-resource environment, an audio-visual interface as well as interface localization can improve the usability, accessibility, and inclusivity of software systems, allowing a broader spectrum of users to effectively connect with the technology despite numerous restrictions and challenges.

Software designers can incorporate these patterns to address the challenges faced by low-resource environments in accessing educational and technological resources. To apply the patterns effectively, the software development process should follow a systematic sequence. First, understand the context of the low-resource environment, identifying specific challenges and target users' language preferences. Next, select the relevant patterns—lightweight development, Audio-Visual Interface, and Language localization—based on the identified challenges. The patterns can be applied individually or in combination. Sequentially apply these patterns during the software development process, optimising resource usage through lightweight development, enhancing user interaction and inclusivity with audio-visual elements, and localising the software interface to the preferred language. Continuously iterate and refine the software system based on user feedback and evolving requirements, ensuring the adopted patterns remain effective and relevant. Thoroughly validate and test the software system to verify successful pattern integration and efficient operation in resource-constrained settings. After deployment, continually monitor the software's performance and user feedback, adapting and optimising the patterns as needed to maintain efficiency and address emerging challenges. By following this approach, tailored solutions for low-resource environments can be developed, delivering accessible, efficient, and user-friendly software systems that effectively address the identified challenges in the target area.

These patterns are similar to approaches being developed for edge computing in the context of networked application design and mobile application development in the context of ICT for development. The specific contexts lead to specialized solutions, such as mobile data management for mobile phones, which is strongly related to more general solutions in low-resource environments. The patterns presented are not restricted to low-resource environments so do not have limited applicability in that sense. They are not complete, however, in that they do not cover all aspects of low-resource environments that influence the design of solutions. For example, implementation

[7]https://cmusphinx.github.io/
[8]https://cloud.google.com/speech-to-text
[9]https://learn.microsoft.com/en-us/azure/cognitive-services/speech-service/overview
[10]https://helpx.adobe.com/xd/get-started.html
[11]https://www.figma.com/
[12]https://www.sketch.com/

approaches such as resource optimization can work in tandem with high-level lightweight design. Further work will explore additional patterns to complement the ones described.

5. CONCLUSION

We have shown how these patterns could be combined and extended (together or in part) into an overall architectural design for software systems in low-resource environments. This we have shown with examples for easy adaptation. Using these patterns can involve a trade-off in robustness and user empowerment for performance or advanced feature sets, but in low-resource environments this trade-off is usually justified and desirable.

REFERENCES

Abou-Khalil, V., Helou, S., Khalifé, E., Chen, M. A., Majumdar, R., and Ogata, H. 2021. Emergency Online Learning in Low-Resource Settings: Effective Student Engagement Strategies. *Education Sciences 11,* 1.

Bon, A., Akkermans, H., and Gordijn, J. 2016. Developing ICT Services in a Low-Resource Development Context. *Complex Systems Informatics and Modeling Quarterly*, 84–109.

Brown, K. and Woolf, B. 2016. Implementation patterns for microservices architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*. PLoP '16. The Hillside Group, USA.

Chavarriaga, E., Jurado, F., and Díez, F. 2017. An approach to build XML-based domain specific languages solutions for client-side web applications. *Computer Languages, Systems and Structures 49*, 133–151.

Di-Pietro, L., Piaggio, D., Oronti, I., Maccaro, A., Houessouvo, R. C., Medenou, D., De Maria, C., Pecchia, L., and Ahluwalia, A. 2020. A framework for assessing healthcare facilities in low-resource settings: field studies in Benin and Uganda. *Journal of Medical and Biological Engineering 40*, 526–534.

Di Pietro, L., Piaggio, D., Oronti, I., Maccaro, A., Houessouvo, R. C., Medenou, D., De Maria, C., Pecchia, L., and Ahluwalia, A. 2020. A framework for assessing healthcare facilities in low-resource settings: field studies in Benin and Uganda. *Journal of Medical and Biological Engineering 40*, 526–534.

Gautam, A., Bortz, W. E. W., and Tatar, D. 2017. Case for Integrating Computational Thinking and Science in a Low-Resource Setting. In *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development*. ICTD '17. Association for Computing Machinery, New York, NY, USA.

Haji, H. A. 2017. Investigating Mobile Graphic-based Reminders to Support Compliance of Tuberculosis Treatment. Ph.D. thesis, Department of Computer Science, Faculty of Science, University of Cape Town.

Idani, A. 2022. A Lightweight Development of Outbreak Prevention Strategies Built on Formal Methods and XDSLs. In *Proceedings of the 2021 European Symposium on Software Engineering*. ESSE '21. Association for Computing Machinery, New York, NY, USA, 85–93.

Jwo, J.-S., Lin, C.-S., Lee, C.-H., and Wang, C. 2021. A lightweight application for reading digital measurement and inputting condition assessment in manufacturing industry. *Mobile Information Systems 2021*, 1–10.

Kowalski, M. J., Elliot, A. J., Guzman, J. C., and Schuenke-Lucien, K. 2022. Early literacy skill development and motivation in the low-income context of Haiti. *International Journal of Educational Research 113*, 101972.

Metatla, O., Oldfield, A., Ahmed, T., Vafeas, A., and Miglani, S. 2019. Voice User Interfaces in Schools: Co-Designing for Inclusion with Visually-Impaired and Sighted Pupils. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Association for Computing Machinery, New York, NY, USA, 1–15.

Meyer, A. J., Armstrong-Hough, M., Babirye, D., Mark, D., Turimumahoro, P., Ayakaka, I., Haberer, J. E., Katamba, A., and Davis, J. L. 2020. Implementing mhealth Interventions in a Resource-Constrained Setting: Case Study from Uganda. *JMIR Mhealth Uhealth 8,* 7, e19552.

Moodley, M. 2020. Localising an online computer software to include setswana, an indigenous african language, for the south african teacher. Ph.D. thesis, University of the Witwatersrand Johannesburg, South Africa.

Ofori-Manteaw, B. B., Dzidzornu, E., and Akudjedu, T. N. 2022. Impact of the COVID-19 pandemic on clinical radiography education: Perspective of students and educators from a low resource setting. *Journal of Medical Imaging and Radiation Sciences 53,* 1, 51–57.

Ramanujapuram, A. and Malemarpuram, C. K. 2020. Enabling Sustainable Behaviors of Data Recording and Use in Low-Resource Supply Chains. In *Proceedings of the 3rd ACM SIGCAS Conference on Computing and Sustainable Societies*. COMPASS '20. Association for Computing Machinery, New York, NY, USA, 65–75.

Rayed, M., Elahi, T., Arefin Shimon, S. S., and Ahmed, N. 2023. MFS Design in Appstore-Enabled Smart Featurephones for Low-Literate, Marginalized Communities. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Association for Computing Machinery, New York, NY, USA.

Rocha, D., Marley, S. A., Drakeford, B., Potts, J., and Gullan, A. 2022. The benefits of guide training for sustainable cetacean-based tourism in developing countries, case study–Ponta do Ouro Partial Marine Reserve, Mozambique. *Journal of Coastal Conservation 26,* 4, 33.

Sahlabadi, M., Muniyandi, R. C., Shukur, Z., and Qamar, F. 2022. Lightweight Software Architecture Evaluation for Industry: A Comprehensive Review. *Sensors 22,* 3.

Scholz, J., Kaspar, J., König, K., Friedmann, M., Vielhaber, M., and Fleischer, J. 2023. Lightweight design of a gripping system using a holistic systematic development process - A case study. *Procedia CIRP 118*, 187–192. 16th CIRP Conference on Intelligent Computation in Manufacturing Engineering.

Sengupta, K., Javeri, A., Mascarenhas, C., Khaparde, O., and Mahadik, S. 2021. Feasibility and Acceptability of a Synchronous Online Parent-Mediated Early Intervention for Children with Autism in a Low Resource Setting During COVID-19 Pandemic. *International Journal of Disability, Development and Education 0,* 0, 1–17.

Shrivastava, A. and Joshi, A. 2019. Directedness and Persistence in Audio-Visual Interface for Emergent Users. In *Proceedings of the 10th Indian Conference on Human-Computer Interaction*. IndiaHCI '19. Association for Computing Machinery, New York, NY, USA.

Suleman, H. 2019. Reflections on design principles for a digital repository in a low resource environment.

Suleman, H. 2021. Simple dl: A toolkit to create simple digital libraries. In *Towards Open and Trustworthy Digital Societies*, H.-R. Ke, C. S. Lee, and K. Sugiyama, Eds. Springer International Publishing, Cham, 325–333.

Suleman, H. 2022. Investigating evolving collection support with simple tools. In *From Born-Physical to Born-Virtual: Augmenting Intelligence in Digital Libraries*, Y.-H. Tseng, M. Katsurai, and H. N. Nguyen, Eds. Springer International Publishing, Cham, 449–455.

Telemala, J. P. and Suleman, H. 2021. Exploring Topic-Language Preferences in Multilingual Swahili Information Retrieval in Tanzania. *ACM Trans. Asian Low-Resour. Lang. Inf. Process. 20,* 6.

Thinyane, H. and Bhat, K. S. 2019. Apprise: Supporting the critical-agency of victims of human trafficking in thailand. CHI '19. Association for Computing Machinery, New York, NY, USA, 1–14.

van Tonder, M., Naude, K., and Cilliers, C. 2008. Jenuity: A Lightweight Development Environment for Intermediate Level Programming Courses. *SIGCSE Bull. 40,* 3, 58–62.

von Holy, A., Bresler, A., Shuman, O., Chavula, C., and Suleman, H. 2017. Bantuweb: A Digital Library for Resource Scarce South African Languages. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists*. SAICSIT '17. Association for Computing Machinery, New York, NY, USA.

Waseem, M., Liang, P., and Shahin, M. 2020. A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software 170*, 110798.