THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Ahuora Adaptive Digital Twin Platform

Ethan MacLeod

Supervised By:
Tim Walmsley, Mark Apperley

Contents

1 Introduction **1**

2 Pinch Analysis **2**
2.1 Historical development of graphical tools 2
2.2 Digital Pinch Tools 3
2.3 Retrofit Vs Design 5

3 Design **7**
3.1 Design process 7
3.2 Design Techniques 8
3.3 UI Evaluation 9

4 Software **12**
4.1 Software Design Principles 12
4.2 Monolithic vs Micro-services 15
4.3 Software technology reviews 16
4.3.1 D3.Js 16
4.3.2 React-vis 17
4.3.3 React-chartjs-2 17
4.3.4 Software technology conclusion 18

5 Summary **19**

6 Appendix **24**

List of Figures

1 Heat Exchanger Network Diagram [1] 2
2 SuperTarget 7 HEN Diagram [2] 3
3 OpenPinch Stream Data Input 4
4 OpenPinch GCC 4
5 Contributions to HEN retrofitting over 30 years [3] 5
6 Wire Frame Tangible vs Digital Results [4] 8
7 A/B Design Process [5] 10
8 Readability Metric [6] 13

List of Tables

1 Stages of UI design and the relative effort allocated to each stage [7] 8
2 API Design Guidelines 14
3 Pros and Cons of D3.Js 16
4 Pros and Cons of React-vis 17
5 Pros and Cons of React-chartjs-2 18

1 Introduction

New Zealand has committed to achieving net zero Greenhouse Gas (GHG) emissions by the year 2050 through their Carbon Change Response Amendment Act 2019 [8]. One of the largest roadblocks to this eventual goal is the industrial process heat sector, which by itself contributes to 28% of all of New Zealand's energy related GHG emissions [9]. Limiting the amount of process heat that needs to be generated will contribute significantly to net zero GHG emissions, while also reducing the money required to generate this heat. One approach to limiting process heat generation is applying pinch analysis technique to industrial process engineering plants across New Zealand. Pinch analysis is a technique that limits the heat required for a system by finding points in which excess heat can be recycled back through the system to heat other processes - effectively reducing the amount of heat that needs to be applied. Unfortunately the current digital tools that exist fail to provide an ideal user experience, and this has led to a low adoption rate across the industry for applying pinch analysis. The aim of this literature review is to identify the shortcomings of these technologies and find appropriate methodologies and principles that would lead to a better user experience overall, in the hopes that following these principles will increase adoption rates.

Some key research questions that will be address in this review are:

1. How are the current digital pinch tools lacking?
2. Which design techniques and processes should be considered when making a pinch tool?
3. What software design principles and practices should be considered when making a pinch tool?

Research into existing pinch tools are included in Section 2 - pinch analysis. This section is a review on the history of graphical representations for process optimization and how these tools are applied to current digital pinch solutions. The most popular digital pinch analysis tools will be scrutinized primarily on their User Interface (UI) and the nuances between design and retrofit in regards to pinch analysis will be discussed. Section 3 - Design focuses on extracting key UI design paradigms and their application to pinch tools. Design principles, ideal design processes, and methods to evaluate the efficacy of designs will be researched and documented in contrast to weaknesses found in existing tools. The software design section - Section 4 - will explore the best practices to consider when designing complex software. Included in this is discussion of software architecture, general software design principles, and a technology review of some applicable graphing tools to process engineering.

One notable exclusion from this document is the principles and processes behind pinch analysis will not be discussed in-depth. The focus of this review is existing pinch tools and ways to improve them, so an in depth discussion of pinch analysis and how it works is out of scope for this document.

2 Pinch Analysis

Pinch analysis is a methodology used in process engineering to optimize energy efficiency within a system [10]. Pinch analysis does this by determining the 'pinch point' - or point of minimum energy efficiency - and using this to inform the placement and interactions between heat exchangers. This section of the report focuses on the development and critique of existing pinch analysis digital tools and graphical representations, as well as their role within process engineering.

2.1 Historical development of graphical tools

Graphical tools have played a significant role in the evolution of process engineering, particularly in enhancing decision-making when employing pinch techniques. Among these tools, several stand out due to their utility and impact:

The Heat Exchanger Network (HEN) diagram is a visual representation of how heat is being exchanged within a system, and includes heat exchangers, streams, and utility heating/cooling inside it [11]. HEN diagrams are used extensively with heat optimization techniques such as pinch analysis, as it provides a clear overview of the heating within a system. Fig. 1 demonstrates the typical layout of a HEN diagram - where the hot and cold streams are shown as the red and blue lines respectively, the heat exchanger of the network are the yellow utilities (Named E(X)) and the heating and cooling utilities are the blue and red utilities (C and H).

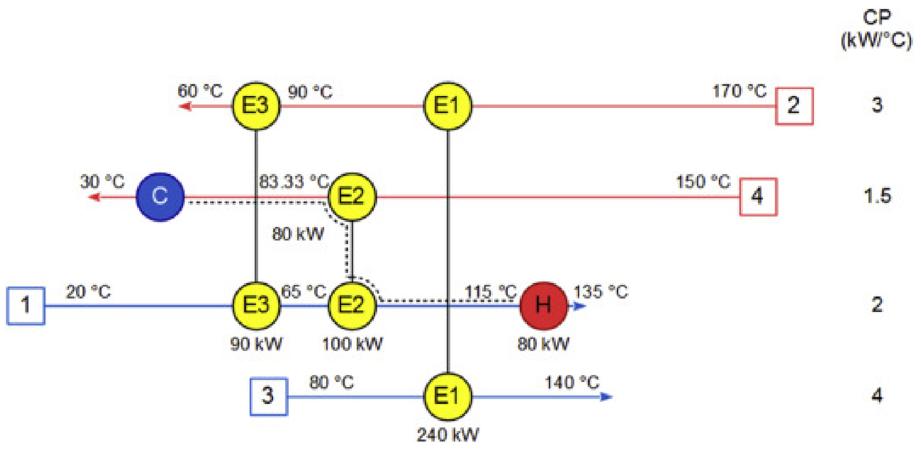


Figure 1: Heat Exchanger Network Diagram [1]

Composite curves, also know as T-H diagrams are used to visualize hot and cold streams on the same graph and also the heat transfer potential between them [12]. A composite curve is used in pinch analysis to identify the pinch point of a system visually, and are instrumental to obtaining the most energy efficient configuration for a HEN - identifying where the minimum temperature difference occurs between streams is a common tactic for targeting with pinch analysis.

Similarly to composite curves, grand composite curves are also used heavily in pinch analysis techniques. A grand composite curve is the graphical representation for how heat flows over temperature intervals - plotted by heating 'duty'. Grand composite curves generally give more information about a system than composite curves and are used to identify heat recovery opportunities within a system.

A Mollier diagram, or Enthalpy - Entropy diagram, was one of the first graphical tools used to represent thermodynamics, more specifically the functional relationship between enthalpy, entropy, temperature, pressure and quality of steam [13]. Adopted in the early 1900s, the Mollier diagram has many applications in process engineering, especially when finding an unknown value in the previously mentioned relationship.

P-graph was another tool developed in the late 1970s as a CAD generation of process structures [14] that was further developed until the main framework for P-graph was finally finished in the 1990s [15]. P-graph uses graphical representation of process networks to optimize and evaluate process structures - and provides optimal alternatives.

Currently there are many tools for visualizing process engineering due to the increase in processing power of modern machines. Programs that provide dynamic simulation and real time data integration are common - such as Aspen Plus, Aspen HYSYS, and DWSIM. These programs can provide outputs utilizing some of the previously mentioned processes while also demonstrating a holistic representation of the entire system.

2.2 Digital Pinch Tools

Some digital tools and libraries have been specifically made integrating pinch analysis techniques. The most popular of these include; SuperTarget [16], OpenPinch [17], and Aspen Energy Analyzer [18]. While these tools address the lack of digital pinch modules, most fall short in terms of their design and usability. This section will be an overview of these tools in terms of their design and functionality.

SuperTarget is a module designed for retrofitting oil refineries and is integrated with the Petro-Sim process simulator. Although it offers high compatibility, allowing users to import files directly from popular process simulators like Aspen HYSYS, Aspen Plus, Petro-Sim, and PRO/II, SuperTarget is often regarded as less relevant in modern settings. This is primarily due to its user interface, which can be considered less intuitive compared to newer tools. Fig. 2 shows the HEN diagram output from SuperTarget 7; the format of the UI is outdated in its design and doesn't provide the breadth of tools needed for interacting with the graph. Despite this, SuperTarget interfacing with other programs allows it to be used without a redesign of the system within the application, meaning it has lots of utility to be used with process simulators that don't have a native pinch module.

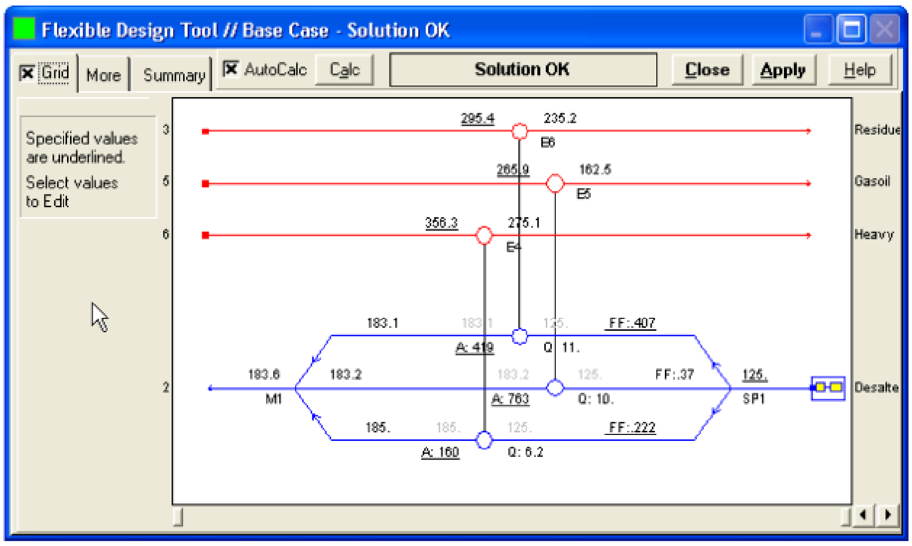
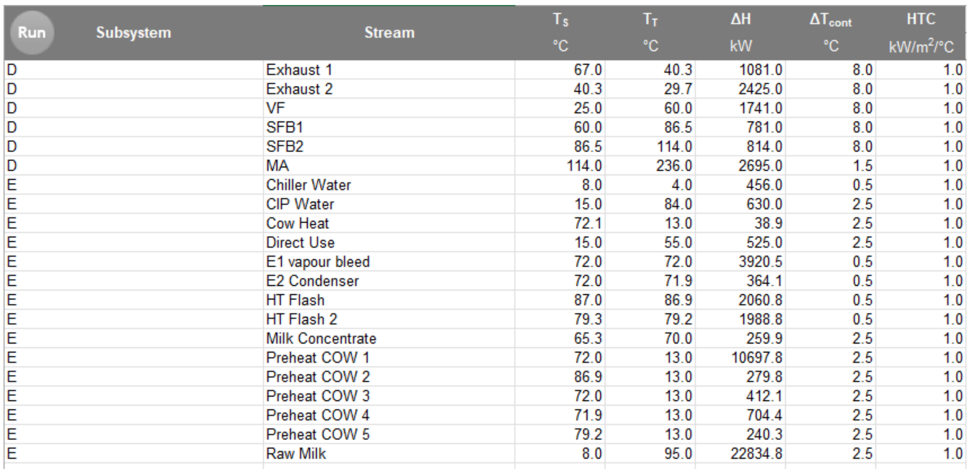


Figure 2: SuperTarget 7 HEN Diagram [2]

OpenPinch is a pinch-based HEN optimizer built on top of Microsoft Excel spreadsheets. Since OpenPinch uses Microsoft Excel and Visual Basic (VB) for its functionality and UI, it is very limited in how it can interact with existing tools. For example, if a user wants to use OpenPinch to evaluate a system made in DWSIM or other process simulators, the user will need to deconstruct the flow-sheet into its base streams then input them manually into the form (Fig. 1). This along with its unintuitive UI makes it hard to navigate for new users, and slows down the workflow of anyone hoping to utilize it. Despite these shortcomings in terms of UI and interfacing with other programs, OpenPinch is among the best pinch-based HEN optimizers for its breadth of functionality and graphical outputs. . In addition to this, Microsoft Excel is known for its visualization of input data, and OpenPinch succeeds at leveraging this to provide clean outputs for graphs such as composite curves and grand composite curves (Fig. 2). Overall OpenPinch places a heavy focus on its functionality while disregarding many UI design principles that it would benefit from, and despite its good output graphs and visualizations, it lacks overall polish.



Run Subsystem	Stream	T_S	T_T	ΔH	ΔT_{cont}	HTC
		°C	°C	kW	°C	kW/m^2/°C
D	Exhaust 1	67.0	40.3	1081.0	8.0	1.0
D	Exhaust 2	40.3	29.7	2425.0	8.0	1.0
D	VF	25.0	60.0	1741.0	8.0	1.0
D	SFB1	60.0	86.5	781.0	8.0	1.0
D	SFB2	86.5	114.0	814.0	8.0	1.0
D	MA	114.0	236.0	2695.0	1.5	1.0
E	Chiller Water	8.0	4.0	456.0	0.5	1.0
E	CIP Water	15.0	84.0	630.0	2.5	1.0
E	Cow Heat	72.1	13.0	38.9	2.5	1.0
E	Direct Use	15.0	55.0	525.0	2.5	1.0
E	E1 vapour bleed	72.0	72.0	3920.5	0.5	1.0
E	E2 Condenser	72.0	71.9	364.1	0.5	1.0
E	HT Flash	87.0	86.9	2060.8	0.5	1.0
E	HT Flash 2	79.3	79.2	1988.8	0.5	1.0
E	Milk Concentrate	65.3	70.0	259.9	2.5	1.0
E	Preheat COW 1	72.0	13.0	10697.8	2.5	1.0
E	Preheat COW 2	86.9	13.0	279.8	2.5	1.0
E	Preheat COW 3	72.0	13.0	412.1	2.5	1.0
E	Preheat COW 4	71.9	13.0	704.4	2.5	1.0
E	Preheat COW 5	79.2	13.0	240.3	2.5	1.0
E	Raw Milk	8.0	95.0	22834.8	2.5	1.0

Figure 3: OpenPinch Stream Data Input

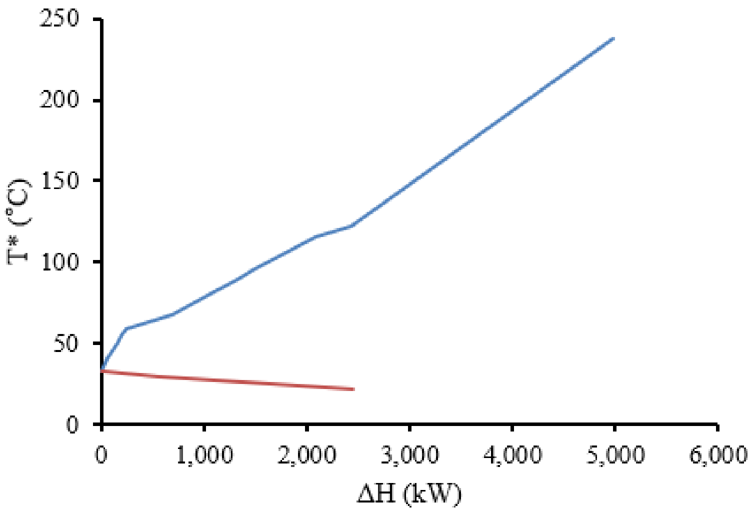


Figure 4: OpenPinch GCC

Aspen Energy Analyzer is the pinch-based HEN module that is integrated with the Aspen process simulation ecosystem. Energy Analyzer as a package is contained with many of Aspen's proprietary software, such as Aspen HYSYS and Aspen Plus - this means for many who use this software, Energy Analyzer is the easiest pinch tool to use. Unfortunately, due to this integration, the UI for Energy Analyzer follows the direction of the software it is contained in - thus it has the same problems of lacking aesthetics and usability. Aspen software can be seen as mostly functional, but not particularly user-friendly or intuitive, especially for new users or those not accustomed to industrial software interfaces. This can lead to a steeper learning curve for navigating and utilizing the software effectively. This lack of attention to streamlining processes and visual indicators is prevalent in the majority of Aspen software. In terms of functionality, Energy Analyzer is considered generally good, however Ben Maes [19] notes in their case study of applying Aspen software to retrofit problems, that Energy Analyzer has a tendency to fail when confronted with complex HEN designs.

Overall, there are many graphical pinch tools that exist for design and retrofit problems, however the majority of these suffer from poor UI and design in general. One of the main benefits of applying pinch methods to HEN problems is being able to visualize the results and make decisions based on them - so usability and design is a major concern for these software. Time To Adopt (TTA) considerations are also lacking in these systems - while these modules are designed for process engineers who already have a fundamental knowledge of the terminology and techniques, this does not necessarily equate to being able to use the software easily.

2.3 Retrofit Vs Design

Ideal application of pinch analysis is done as a 'blank sheet' design - where the design for the HEN is built from the ground up [10]. Unfortunately in many cases this isn't realistic, and frequently existing systems need to be changed and optimized - this process is known as 'retrofit' or 'retro-vamp'. Truls Gundersen [20] estimates that roughly 70% of all projects in the process sector are for retrofitting existing systems, which shows the importance of this technique. While both of these approaches aim for the same end goal (optimization of a HEN) there are some nuances in the approaches of each. Sreepathi et al [3] in their study of retrofitting methodologies notes their findings of the evolution of retrofitting techniques over time (Fig. 5).

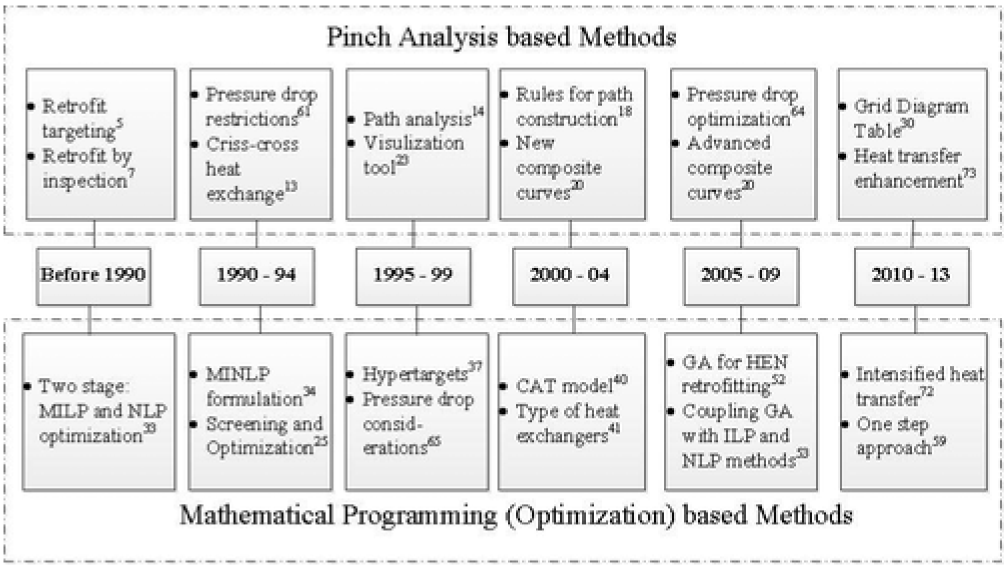

Pinch Analysis based Methods
• Retrofit targeting[5] • Retrofit by inspection[7]
• Pressure drop restrictions[61] • Criss-cross heat exchange[13]
• Path analysis[14] • Visulization tool[23]
• Rules for path construction[18] • New composite curves[20]
• Pressure drop optimization[64] • Advanced composite curves[20]
• Grid Diagram Table[30] • Heat transfer enhancement[73]
Before 1990
1990 - 94
1995 - 99
2000 - 04
2005 - 09
2010 - 13
• Two stage: MILP and NLP optimization[33]
• MINLP formulation[34] • Screening and Optimization[25]
• Hypertargets[37] • Pressure drop consid-erations[65]
• CAT model[40] • Type of heat exchangers[41]
• GA for HEN retrofitting[52] • Coupling GA with ILP and NLP methods[53]
• Intensified heat transfer[72] • One step approach[59]
Mathematical Programming (Optimization) based Methods

Figure 5: Contributions to HEN retrofitting over 30 years [3]

HEN retrofit can be broadly classified into two groups: thermodynamic-mathematical methods and pure mathematical programming methods [21]. Thermodynamic-mathematical methods, typically involving pinch analysis, employ various graphical analytical tools previously discussed, such as Grid Diagrams (GD), Composite Curves (CC), and Grand Composite Curves (GCC) for retrofitting. The pinch concept in retrofit methods also incorporates mathematical programming techniques in later stages to optimize the network design. These techniques often aim to minimize costs and enhance energy efficiency, considering the existing constraints of the heat exchanger network (HEN).

Pure mathematical programming based methods - also called optimization methods - can be further reduced to two main processes; deterministic and stochastic optimization. Deterministic optimizations the technique that is employed when all variables and states are known - IE when there are no degrees of freedom in a system, and utilizes linear and non-linear algebra [22]. Stochastic optimization on the other hand supports many unknowns and is more likely to find the global optimum for HEN retrofit problems [23]. Despite these differences, some variations on these methods exist that combine both approaches, such as that proposed by Zhou et al [24] in their case study of a hybrid stochastic-deterministic method for process optimization.

From a graphical tool point of view, retrofitting an existing system requires extra functionality that includes its own design and considerations. The three tools mentioned previously all contain some retrofitting capabilities; OpenPinch requires manual inputs while SuperTarget and Aspen Energy Analyzer interface with their respective process simulators.

3 Design

As discussed in Section 2.2.2 there is a general lack of focus on design principles and usability in existing pinch graphical tools. OpenPinch, for example, is a purely functional program that fulfills all the practical needs of a pinch analysis system while existing in Microsoft Excel sheets. The lack of usability and unintuative design harms the user experience and leads to a system that is hard to navigate and requires a much steeper learning curve. In an effort to demonstrate which principles and methodologies drive great UI and design, this section will discuss the process and considerations that need to be made. Included in this is the general approach of the design process, some key design techniques that are used, and methods to evaluate the efficacy of a UI design.

3.1 Design process

The standard procedure for many large web applications has been that the majority of the UI design is compiled upfront - usually 85% - 95%, then as usability and acceptance testing is conducted, changes are made and drive the development of the project [25]. This process relationship between development and design often comes at the cost of the design iterations failing to keep up with the fast-paced coding sprints found in Extreme Programming (XP) and inevitably leads to a decline in actual development. The solution is thus to have the design process be dependant on what work has been done in the last agile development sprint - 'Development iterations drive usability testing' and then the next development sprint reflects the changes in requirements from usability and acceptance testing - 'Usability Testing Results in Changes in Development'. Using agile project management techniques that integrate both the design and development leads to a symbiotic relationship between these two often separate aspects of a project and allows the requirements of the project to smoothly change as testing is conducted.

One notable limitation of this design pattern is how it is generally tailored for large development teams with separate UI/User Experience (UX) and programming teams. For smaller teams or individual developers, considerations have to be made as to where the majority of the workload will lie - since after applying this methodology each sprint now has two dependant factors, usability testing and development. Existing research is limited on how this development style affects teams of varying sizes and warrants further investigation.

McInerney et al [7] in their case study of common problems that can be identified in many design projects, designed a metric for where efforts should be focused during a design process (Table 1). The design phases from Table 1 can be generalised into key terms: UI architecture is wire-framing and low fidelity designs, UI detail design is prototyping and high fidelity models, and finally UI design change is the process of editing the designs to conform to results from user testing and evaluation. The data indicates that high fidelity models demand the most effort, primarily due to extensive interactions between designers and developers. This study was conducted on many UI professionals and developers to find these values, however in this study no considerations were made to the specifics of the projects and experiences these participants had. Some further research that can be done is to match these values to specific projects - like web design, application design etc - and see how the project determines where the emphasis on effort is placed.

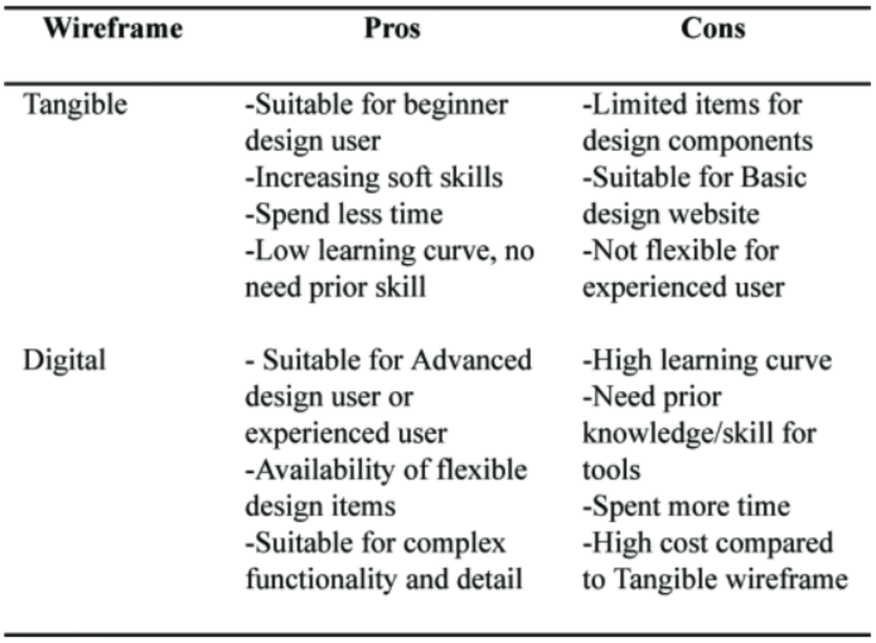
UI Design Phase	Description	Effort
UI architecture	Defining the UI at a gross level, defining the key design direction	20 - 30%
UI detail design	Defining the UI at a level detailed enough to serve as instructions for the programmers	50%
UI design change management & verification	Design rework to address issues that arise after the UI design freeze; also includes work to verify that the UI was implemented as specified	20 - 30%

Table 1: Stages of UI design and the relative effort allocated to each stage [7]

3.2 Design Techniques

During the design process of any large system, there are two key milestones - wire framing and prototyping. Each of these techniques serves their own purpose despite their similarities in execution, and where to use each of them is an important skill for UI/UX developers to learn.

Wire framing is a valuable development tool that interfaces very well with agile management techniques - it involves creating low-fidelity visual representations of UI systems, typically only roughly based on the requirements of the system [26]. Wire frames can be done quickly while still providing valuable information on UI design, and works best after 'round 1' research, where key users and requirements have already been identified and developers know the design metrics [27].



Wireframe	Pros	Cons
Tangible	-Suitable for beginner design user -Increasing soft skills -Spend less time -Low learning curve, no need prior skill	-Limited items for design components -Suitable for Basic design website -Not flexible for experienced user
Digital	- Suitable for Advanced design user or experienced user -Availability of flexible design items -Suitable for complex functionality and detail	-High learning curve -Need prior knowledge/skill for tools -Spent more time -High cost compared to Tangible wireframe

Figure 6: Wire Frame Tangible vs Digital Results [4]

Sutipitakwong et al [4] notes in their study of physical wire framing vs digital wire framing tools (Fig. 6), that tangible wire frames are superior when developing low-tech quick iterations of a design, while digital wire frames can provide much more upfront functionality and can lead straight into prototyping. While this study manages to find some of the strengths and weaknesses of these wire framing styles, continued research into the differences that can be found between specific digital tools and wire framing methods could serve to quantify the differences further.

Prototyping on the other hand is an approximation of a final design or concept [28]. In contrast to wire framing, prototyping is used much later in the design process, and typically contains far more functionality that is representative of the final system. By making a design that emulates a system without the majority of the infrastructure required for the final implementation of that system, testing becomes easier and issues that cannot be found in a lower fidelity systems can be identified [29]. There are many different techniques and tools that can be utilized when prototyping a system, and choosing the best tool is left to the requirements of the project.

3.3 UI Evaluation

An important portion of the design process is being able to find key metrics and techniques for evaluating the efficacy of a UI design [30]. When evaluating a UI system, the standard testing methods usually fall into two main categories: empirical and inspection [31]. Heuristic evaluation involves experts, or UI professionals, navigating through a user interface identifying problems and pain points, while user testing is done by stakeholders and end users in the same process. Nielson et al [32] created the ten metrics for heuristic testing in 1990, and many other professionals have since expanded on these principles to layout a solid framework for conducting heuristic testing - such as Murillo et al [33] and Allen et al [34].

In general, when conducting usability testing on a system for both empirical and inspection, the most important methodology for gaining valuable feedback is controlling the social dynamics with the test subjects. Carol Barnum [35] notes the main techniques for gaining impartial results - most of these are ways to ensure that the tester is not directly or indirectly affecting the testee. These behaviours can include:

1. **Neutral Prompts** - A prompt to the user should not direct them in what to do.
2. **Comfortable Testing Environment** - The user should feel comfortable to get results that are indicative of their regular behaviour.
3. **Unbiased responses** - A tester should give little to no feedback on users interactions, but where necessary the responses should be unbiased.
4. **Balance Praise** - A tester should not give excessive praise towards user behaviours.

A novel way of utilizing heuristic testing methods that additionally categorizes the findings was presented in the work of Murillo et al [33] in their case study. The first step in this method is to gather some professionals and have them conduct their own individual usability evaluation on the system. This is done to ensure that each participant has the opportunity to express their thoughts without interference from other participants. The next step is to gather the participants and have them unify their findings into a single list following which the the participants are asked to cooperate in ranking the issues based on a metric determined by the test conductor. After all these steps the test conductor will have a ranked list of issues in order of their severity. This method of heuristic testing is ideal when a lot of time can be allocated to the testing phase, but this needs to be determined and used selectively. One noticeable lack of discussion in this study regarding this method of heuristic testing is how it compared to the standard heuristic testing philosophies, as the heuristic testing technique potentially sacrifices post-test processing in favour of a higher upfront time cost of setting up the evaluation. An evaluation of how this style of testing compares to other methods when considering the additional time required could be beneficial to consolidate this method as legitimate - a time to quality/number of results graph in particular would be beneficial.

A/B testing is another common testing technique found in the industry as noted by Siqueira et al [36]. A/B testing is the process in which two iterations of a design are presented to a test participant and each are evaluated on their own merits. Fig. 7 notes the general process in which this is done from . First, Two variants of a design are made and a hypothesis is determined on which design will perform better under predetermined testing requirements. Second, multiple users are given a random variant and asked to walk through the system through prompts. Finally the data from the experiments are compiled and compared against the hypothesis. A/B testing typically gathers hard empirical results, such as clicks and time spend on UI elements as well as some user engagement metrics, which differs itself from the other testing methods which are typically more subjective and generally contain a lot more stylistic pain points. This also then requires some statistical post-processing of the data to ensure that differences are significant. Another benefit from A/B testing is the 'better' variant will be used in the next testing cycle. There are many A/B testing frameworks that can be used, such as the one outlined by Siqueira et al [36], but these general frameworks often require fine tuning based on what the tester is hoping to achieve from their hypothesis as well as which specific parts of a system they are focusing on. Finding the best metrics for a given project is something that is glossed over in Quin et al [5], but this seems to be very important in generalizing methods for A/B testing.

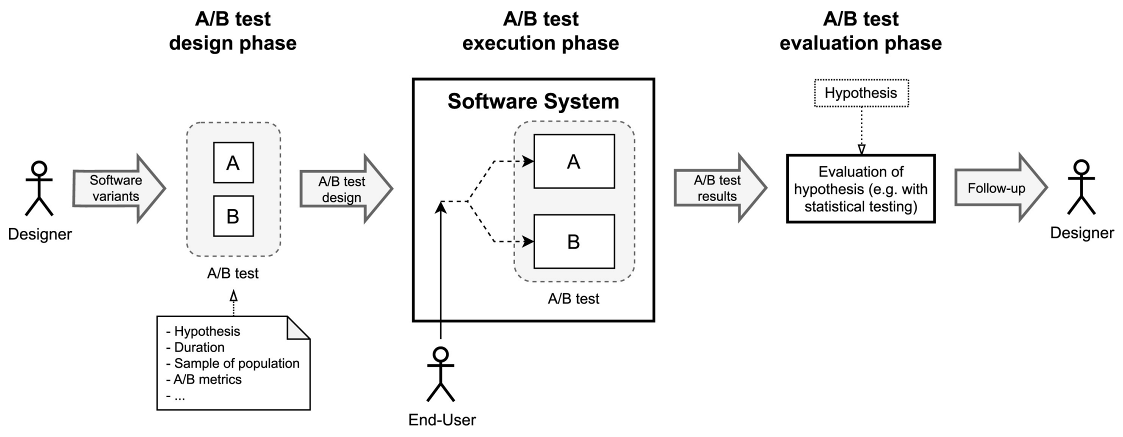


Figure 7: A/B Design Process [5]

The difference in results from heuristic and user testing were identified in a case study presented by Wang et al [37]. The results from the heuristic group demonstrated that many more potential pain points and issues can be identified by UI professionals (an average of 20), but these issues contained a higher than average set of 'false positives' (false alarms) and minor issues that don't have a large affect on how the user interacts with the system. The user testing group however had a much smaller number of identified pain points, but each of these - by definition - are those that needed to be addressed. The difference then can be seen that user testing does well to identify large 'real' pain points that should take priority in the next design and development cycle while heuristic testing are typically contain a large number of minor fixes that should be blocked until extra development resources can be allocated. Both of these methods have their own merits and should be used at the conductors discretion.

In general finding a balance of both empirical and inspection testing is ideal for evaluating a UI system. Both of these methods provide their own valuable points of feedback and favouring one over the other can lead to misidentified or unimportant testing results. In addition to this, the specific process of empirical or inspection testing should be determined by the scope and time constraints of the testing process. This section has identified two general methods that fit into these categories - but depending on the project more specific evaluation methods may perform better. For example a pinch analysis tool may benefit from additional user testing with process engineers, and have a novel user testing technique based around the workflow of a process simulator.

4 Software

In the realm of process engineering - and specifically pinch analysis techniques - identifying appropriate software tools is crucial. Pinch analysis lends itself well to digital solutions, as pinch analysis uses many Mixed-Integer Linear Programming (MILP) standards. This section analyses various software and software principles through the lens of finding best practices in the development of these systems to achieve usable and intuitive software. Being able to identify the metrics through which high quality software is produced is important during both the development of software and the evaluation of tools to be used in development.

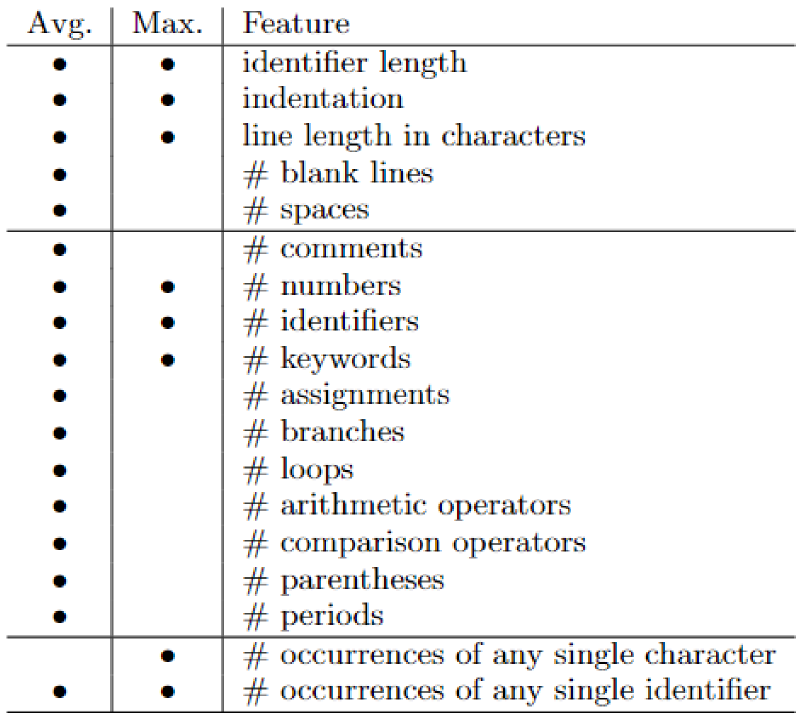
4.1 Software Design Principles

J. R. Mckee [38] estimates that roughly 65%-75% of total support and software development efforts are spent on the maintenance of a system. This is largely due to the lack of frameworks available for measuring the maintainability of a system - and without these metrics it becomes much harder to identify which modules and processes need more considerations for their future maintenance. Aggarwal et al [39] proposes the following three metrics to identify the maintainability of a software system:

1. Readability of source code (RSC)
2. Documentation Quality (DoQ)
3. Understand-ability of Software (UoS)

These key points were identified to have the most impact on maintainability from the stand point of future development of a system - instead of bug fixed and optimizations. RSC is the measure of comments and file structure of the source code, in this case study it was [$NumLinesCode/NumLinesComments$]. The second metric, DoQ measures the quality of documentation of the system using Gunning's Fog Index [40]. The final criteria is a combination of the other two, UoS is how well the documentation (DoQ) equates to the source code (RSC) via how its 'language' is similar between them.

One of the main concepts used by Aggarwal et al [39] to evaluate the maintainability of a system was readability. Readability can be defined as the human judgement of how easy a text is to read - in software engineering principles it applies to code instead [41]. Readability is a design principle that should be prevalent in every section of a application, and is almost always used to facilitate other principles such as maintainability, usability, and general code quality. It's due to this that it becomes important to be able to measure the readability of a system consistently. One of the most popular frameworks for evaluating the readability of a piece of software was captured by Buse et al [42] as Fig. 8 - this framework is an empirical measure of readability and can be automatically calculated by tools that gather and compile the results, making it ideal for CI/CD pipelines. An interesting future research point that could be made is how these readability metrics can be affected or improved by the use of AI. The development or research into an AI tool that can automatically improve the readability of a piece of software would be very valuable, and the discussion around whether it performs better than traditional tools (like those based on Gunning's fog index) could be done.



Avg.	Max.	Feature
•	•	identifier length
•	•	indentation
•	•	line length in characters
•		# blank lines
•		# spaces
•		# comments
•	•	# numbers
•	•	# identifiers
•	•	# keywords
•		# assignments
•		# branches
•		# loops
•		# arithmetic operators
•		# comparison operators
•		# parentheses
•		# periods
	•	# occurrences of any single character
•	•	# occurrences of any single identifier

Figure 8: Readability Metric [6]

Another powerful software design principle is Usability - especially in the case of software development tools. Usability as a design principle is generally thought of as secondary to functionality when developing an Application Programming Interface (API) for any type of service - especially in the case that an API is being developed under agile project management styles, where Minimum Viable Product (MVP) development is often preferred. Myers et al [43] challenge this style of API development due to the fact that usability should be the primary focus for an API, as an API at it's core is built around the interactions it will have with users - both developers and stakeholders. Typically, the goal of any API developer is to minimize support and development costs, reduce deployment time, and maximize adoption, especially for public APIs. On the other hand, developers looking to adopt an API focus on simplicity, consistency, and reliability. Often these requirements conflict and lead to an API that is difficult to consume, or in some cases lead to many production issues when developers cannot use an API in its intended way.

The solution to mitigating these potential issues is to place a heavy focus on the design of the API before its implementation. Taking extra time to conduct usability testing with developers can ensure that the structure of an API follows the 'mental model' that a developer will have when consuming it - this includes components such as class names, endpoint structures, and default values of objects. Having an API that is intuitive to use decreases adoption time, as shown by Stylos et al [44], development time can be decreased from 2.4 - 11.2 times and since productions roll out times will be faster and less prone to bugs, users will also be positively affected.

API development in general conflicts with agile development techniques, as retroactive changes to an API that is in production will lead to legacy code breaking down and familiarity developers have with the API degrading. This demonstrates the importance of appropriate API design before any development iterations. When adoption time for a certain API is too large for a project, other alternatives are available to switch to - this is not the case for internal APIs or niche APIs with no alternatives. While there are many ways one can evaluate the usability of an API, as from Gill et al [45], there is still a overarching lack of systematic patterns that lead to a usable API - a gap that will need to be filled.

When evaluating how usable an API is, Grill et al [45] used the following metric in their case study:

Name	**Description**
Complexity	An API should not be too complex. Balance complexity and flexibility. Use abstraction.
Naming	Names should be self-documenting and used consistently.
Caller's Perspective	Make the code readable, e.g., `makeTV(Color)` is better than `makeTV(true)`.
Documentation	Provide documentation and examples.
Consistency and Conventions	Design consistent APIs (order of parameters, call semantics) and obey conventions (get/set methods).
Conceptual Correctness	Help programmers use an API properly by using correct elements.
Method Parameters and Return Type	Use few parameters. Return values should indicate the result of the method. Use exceptions when exceptional processing is needed.
Parameterized Constructor	Always provide a default constructor and setters rather than a constructor with multiple parameters.
Factory Pattern	Use the factory pattern only when inevitable.
Data Types	Choose correct data types. Avoid forcing users to use casting. Do not use strings if a better type exists.
Concurrency	Keep concurrent access in mind.
Error Handling and Exceptions	Define class members as public only when necessary. Handle exceptions near where they occur. Error messages should convey sufficient information.
Leftovers for Client Code	Minimize the amount of code the user needs to type.
Multiple Ways to Do One	Do not provide multiple ways to achieve the same thing.
Long Chain of References	Avoid using long complex inheritance hierarchies.
Implementation vs. Interface	Prefer interface dependencies as they are more flexible.

Table 2: API Design Guidelines

Similarly to how Myers et al [43] outlines the success metric when designing the API, Grill et al's [45] criteria primarily revolves around matching potential developer's mental 'image' of how the API will be set out and ensuring default values are consistent. Method parameters, parameterized constructor, data type, concurrency, multiple ways, and interface implementations are all examples of applying this methodology.

4.2 Monolithic vs Micro-services

The complexity of a system has a large impact on the maintenance time and cost - Banker et al [46] found in their study evaluating software systems based on metrics such as number of modules and module size that maintenance costs can be raised up to 25% when a system is too complex. From this, the software engineering has two general approaches to developing complex systems; monolithic and micro-services.

A monolith is a traditional project style, where a system shares a single code base and typically none of the internal modules are independently executable [47]. All of the functionality, components, and UI elements are contained within a single program platform - which generally make them easier to deploy, develop and test [48]. These benefits make it easy for small to medium size projects, where large scaling and maintenance aren't a factor - but as the complexity of a monolithic program increases, there is a sizeable increase in resource consumption. On the other hand, micro-services are designed to separate concerns into independent modules [49]. The concept of micro-services is not new, however with the industry shift to cloud base approaches, micro-service design has become increasingly popular. Some of the benefits of micro-service design include vast scalability, easy error isolation, and a positive cost-performance return.

An increasingly popular software design architecture that is being adopted by many large tech companies is the 'Modular Monolith' [50]. This architecture is a type of monolithic application that is internally structured into loosely coupled modules. This approach combines the straightforward deployment benefits of a monolithic system with the maintainability and separation of concerns usually associated with a modular system. Although modular monoliths can serve as a standalone architectural choice, they also offer a potential stepping stone to micro-services architectures, making it easier to decouple these individual modules at a later date to operate as their own services. The utility of a modular monolith isn't limited to large tech companies with numerous services that need to be released. Often, when developing a complex system, a modular monolith can provide a scalable yet manageable framework that supports a range of functionalities in a unified environment. This architectural style is particularly beneficial in scenarios where teams are transitioning from a traditional monolithic approach and are not yet ready to adopt a fully distributed micro-services architecture. Additionally, modular monoliths can reduce the complexity and overhead associated with deploying and managing multiple independent services while still allowing for clear boundaries and independence between different functional areas [51]. This makes them ideal for projects where rapid scaling or iterative testing of isolated features is necessary.

The choice of system design process is dependant on the project and needs to be evaluated on a case-by-case basis. For small to medium-sized projects, monolithic and modular-monolithic architectures are generally best, as these projects do not require extensive scalability considerations in the short term. For these projects the ease of deployment, testing, and maintenance is the driving factor for choosing monoliths. When a project is identified to require high scaling in the future, or single modules of a system need to be accessed independently of the entire system, micro-service design is superior. For example a process simulator may benefit from following a micro-services design, where a pinch module is included separately with the functionality of interfacing with other simulators.

One key advantage micro-service design also has over a monolith is how well it lends itself to distributed systems and computing [52]. In the pinch tools previously mentioned, Energy Analyzer and SuperTarget were a part of the monolithic designs of their process simulator - they could not be used independently without them. This integration, while easy, restricts flexibility and scalability that could be enhanced through a micro-services architecture.
A micro-services approach to developing pinch analysis tools could significantly leverage distributed programming to enhance performance and scalability. In a distributed system, different services can run on various servers or even across multiple data centers, allowing for more efficient data processing and management. For pinch analysis, this means that complex calculations for energy optimization, such as the generation and manipulation of composite curves and grand composite curves, could be handled by separate, specialized services. Each service could be optimized for specific tasks like data collection, heat integration analysis, or optimization routines, and scaled independently according to demand.

4.3 Software technology reviews

One intersection between software and pinch analysis techniques is graphical representations. Pinch analysis produces many visual outputs that are used to refine and further optimize HEN networks - and its due to this reason that it becomes important to find software tools that can produce accurate graphs and diagrams for to match these outputs. This section is a review of the most popular React frameworks that support graph creating and management in an effort to find an ideal library for pinch analysis.

4.3.1 D3.Js

D3 [53] is a JavaScript framework for creating interactive data visualizations, and is largely used for displaying large complex charts and graphs. Due to it's high skill ceiling, D3 is among the best when creating custom visualizations for web projects. Table 3 contains the notable pros and cons of using D3.Js in a project.

	Pros	**Cons**
1	High level of customization allows for detailed and tailored visualizations.	Steep learning curve, especially for those not familiar with modern JavaScript and SVG.
2	Strong community support and documentation	The frequent updates and changes in D3.Js versions can make it difficult to find current examples or tutorials that match the latest API.
3	Performant with small-medium datasets	Performance degrades with very large datasets (Like those for machine learning)
4	Integrates well with other libraries	Lacks high level components

Table 3: Pros and Cons of D3.Js

D3's main selling point is the customizability and complexity of designs it can make. This makes it ideal for projects that have unconventional graphs, or need more functionality than other graphing libraries would allow the user to create. This complexity along with the extensive community support, official documentation, and performance are the main pro's that come along with using D3. Some notable limitations of the D3 library, included in the cons section of 3, are the steep learning curve, lack of high level components, and the constant updates. A side effect of D3's high skill ceiling is it has a very steep learning curve - all of the functionality that D3 offers means that it takes much longer for beginners to learn. Additionally D3 does not package high level components into its library - components such as default graphs and structures need to be created manually by the developer. D3 is a library that has continuously been updated over its 13 years since creation - its because of this that many examples online of D3 projects are outdated and no longer use the current best practices. Overall, D3 is an incredibly powerful graphing framework that comes with the caveat of requiring much more effort and learning than other low-level libraries that place a greater focus on ease of use. Custom graphs such as HEN diagrams is where D3 excels, one such example from Mike Bostock [54] demonstrates this very well - an interactive graph with changing data points and a lot of complexity.

4.3.2 React-vis

React-vis [55] is a React-based data visualization library built on top of D3 and developed by Uber [56], designed to simplify the process of creating charts and graphs. React-vis offers a variety of chart types such as line, bar, scatter, and pie charts, which are optimized for use within React environments.

	Pros	**Cons**
1	Simple API that allows for quick development.	Not much functionality for custom graphs.
2	Strong integration with the React ecosystem.	Lacks depth of graph features.
3	Maintained frequently by Uber.	Performance degrades with very large datasets.
4	Shallow learning curve.	Poor interactivity with graphs.

Table 4: Pros and Cons of React-vis

The strengths of React-vis can also be seen as the majority of its weaknesses. React-vis is primarily designed for fast development, with many React components available to simply add data to and use. This focus leads to a library that lacks customization options and complexity, making it less than ideal for any project that requires a complex graph, but great for projects that just need the graphs that they provide. This is in stark contract to other libraries such as D3 which places a heavier focus on broad functionality of graphs. Table 4 demonstrates this the best - where most of the pros revolve around how many pre-built components that are packaged with React-vis, and the cons are mostly about how its lacking in custom functionality. This lack of custom functionality means that for building HEN and other system diagrams, React-vis falls short for whats needed. For some graphs such as CCs and GCCs React-vis would be an ideal library - as these graphs are simple outputs plotted on a Cartesian plane and this is what React-vis excels at - but React-vis does not have any pre-built HEN diagrams included in it's package.

4.3.3 React-chartjs-2

React-chartjs-2 (RC2) [57] is a React wrapper for the popular JavaScript library, ChartJs [58]. ChartJs is a library that makes it very easy to create simplistic graphs - components such as bar, line, and pie charts are baked into the package and can be easily added to projects. RC2 has

	Pros	Cons
1	Rich visualization options for included graphs.	High learning curve for those now familiar with React.
2	Strong integration with the React ecosystem.	Large overhead and package size.
3	Responsive and supports interactivity.	Limited to ChartJs functionality.
4	Strong community support.	Performance degrades with very large datasets.

Table 5: Pros and Cons of React-chartjs-2

the exact same functionality as ChartJs but is instead designed to be used with React projects. Some of the pros and cons of RC2 are included below in Table 5.
RC2 contains a lot of the same issues found with D3 and React-vis, with degrading performance with large datasets and package size and because RC2 is based on the ChartJs library, all of the functionality that RC2 has is limited to that offered in ChartJs. RC2 has some notable advantages to the other libraries discussed however - ChartJs has existed much longer than other libraries and has much more community support and documentation, and RC2's backwards compatibility with ChartJs means that it benefits from this as well. RC2 also has much of the same capabilities as D3 in its interactive graphing and responsiveness. The lack of custom graphs is again an issue with this framework, similarly to React-vis and is thus not suitable for building HEN diagrams.

4.3.4 Software technology conclusion

From the reviews conducted on the three frameworks available, D3 is the ideal library to be used in conjunction with pinch analysis. React-vis and RC2 both suffer from their lack of support with custom graphs and data structures - meaning that these libraries cannot be used for the entirety of the pinch system and additional libraries will need to be used to fill this gap. Using D3 would mean a higher workload in general - as D3 doesn't natively include support for popular high level graphs that will need to be used in a pinch system and the overall learning curve for D3 is much higher than the other two solutions. Considerations will also have to be made when using D3 to ensure that large amounts of data aren't being used for graphing when they aren't needed - as there is a sharp decrease in performance with larger amounts of data - however this is a common problem between most React graphing libraries and can't be fixed by finding a better solution.

5 Summary

In conclusion, this literature review has examined existing pinch analysis tools through the lens of usability, design, and software principles. The aim of this report has been to note some issues with current pinch analysis tools, then explore the principles and techniques that would produce a higher polished system. First, the current state of pinch tools were examined, along with some historical context of the progression of graphical process engineering methods. From this, pinch analysis tools were found to be generally lacking in usability and intuitive design, with most of the popular pinch tools having an outdated or minimal UI.

In the design section, some key design principles were explored within the context of applying them to these pinch tools. The design process to follow, the techniques in which to develop high-quality user interfaces, and common methods of evaluating the UI system were found. Wire framing and prototyping were the key techniques that these systems would benefit from and testing methods such as heuristic and A/B testing were explored.

Next, software principles that indirectly contribute to the user experience, such as maintainability, usability, and scalability were identified within the confines of the project, while graphing libraries to facilitate the graphical outputs of the system were evaluated against each other.

Conclusions drawn from this literature review will be used to inform decisions around the development of a pinch analysis module in the future. The specific techniques around implementing maintainability and a usable API will be applied in the development stage, while design paradigms and evaluation methods will drive the front-end UI development cycle. In addition to this, the current issues regarding existing pinch tools will be noted and a conscious effort will be made to avoid these pitfalls.

Considering the limitations of current tools regarding scalability, the new pinch analysis module will be designed as a micro-service. This approach will allow it to leverage the benefits of an existing distributed system, enabling it to function efficiently as an independent module within larger systems. This scalability is crucial for accommodating varying load demands that current tools don't have to address, and integrating seamlessly with existing and future technologies in a process simulator.

References

[1] Jun Yow Yong, Petar Sabev Varbanov, and Jiří Jaromír Klemeš. Heat exchanger network retrofit supported by extended grid diagram and heat path development. *Applied Thermal Engineering*, 89:1033–1045, 2015.

[2] KBC Advanced Technologies. Supertarget 7 release notes. `https://www.kbcat.com/php/sd/ST7-Release%20Notes.pdf`, 2023. Accessed: 2024-04-29.

[3] Bhargava Krishna Sreepathi and G. P. Rangaiah. Review of heat exchanger network retrofitting methodologies and their applications. *Industrial Engineering Chemistry Research*, 53(28):11205–11220, 2014.

[4] Sutipong. Sutipitakwong and Pornsuree. Jamsri. Pros and cons of tangible and digital wireframes. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–5, 2020.

[5] Federico Quin, Danny Weyns, Matthias Galster, and Camila Costa Silva. A/b testing: A systematic literature review. *Journal of Systems and Software*, 211:112011, 2024.

[6] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 73–82, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Paul McInerney and Rick Sobiesiak. The ui design process. *SIGCHI Bull.*, 32(1):17–21, jan 2000.

[8] New Zealand Government. Climate change response (zero carbon) amendment act 2019. `https://environment.govt.nz/acts-and-regulations/acts/climate-change-response-amendment-act-2019/`, 2019. Accessed: 2024-04-29.

[9] Innovation New Zealand Ministry of Business and Employment. Process heat current state fact sheet. `https://www.mbie.govt.nz/dmsdocument/152-process-heat-current-state-fact-sheet-pdf`, 2022. Accessed: 2024-04-29.

[10] Ian C. Kemp. *Pinch analysis and process integration : a user guide on process integration for the efficient use of energy.* Butterworth-Heinemann, Amsterdam ;, 2nd ed. edition, 2007.

[11] Natchanon Angsutorn and Rungroj Chuvaree. 24th european symposium on computer aided process engineering. In *Computer Aided Chemical Engineering*. Elsevier, 2014.

[12] Assaad Zoughaib. *From Pinch Methodology to Pinch-Energy Integration of Flexible Systems.* ISTE Press - Elsevier, 2017.

[13] S. Bobby Rauf. *Understanding Mollier Diagram*, page 8. River Publishers, 2 edition, 2023.

[14] F. Friedler, T. Blicket, J. Gyenis, and K. Tarjáns. Computerized generation of technological structures. *Computers Chemical Engineering*, 3(1):241–249, 1979.

[15] Ferenc Friedler, Kathleen B Aviso, Botond Bertok, Dominic CY Foo, and Raymond R Tan. Prospects and challenges for chemical process synthesis with p-graph. *Current Opinion in Chemical Engineering*, 26:58–64, 2019. Energy, Environment Sustainability: Sustainability Modeling Reaction engineering and catalysis: Green Reaction Engineering.

[16] KBC Global. Supertarget 7, 2023.

[17] Tim Walmsley. Openpinch. Open Source Software, 2024.

[18] AspenTech. Aspen energy analyzer. Software available from Aspen Technology, Inc., 2023.

[19] Ben Maes. *Exploring Aspen Energy Analyser to Improve Processes.* Master's thesis, Instituto Superior Técnico, June 2018. Thesis to obtain the Master of Science Degree in Chemical Engineering.

[20] T. Gundersen. *Retrofit Process Design - Research and Applications of Systematic Methods*, pages 213–240. CACHE Elsevier, 1990.

[21] Ming Pan, Igor Bulatov, and Robin Smith. Heat transfer intensification for retrofitting heat exchanger networks with considering exchanger detailed performances. *AIChE Journal*, 64(6):2052–2077, 2018.

[22] A. R. Ciric and C. A. Floudas. A retrofit approach for heat exchanger networks. *Computers & Chemical Engineering*, 13(6):703–715, 1989. Accessed through SciFinder.

[23] Robin Smith, Megan Jobson, and Lu Chen. Recent development in the retrofit of heat exchanger networks. *Applied Thermal Engineering*, 30(16):2281–2289, 2010. Selected Papers from the 12th Conference on Process Integration, Modelling and Optimisation for Energy Saving and Pollution Reduction.

[24] Teng Zhou, Yageng Zhou, and Kai Sundmacher. A hybrid stochastic–deterministic optimization approach for integrated solvent and process design. *Chemical Engineering Science*, 159:207–216, 2017. iCAMD – Integrating Computer-Aided Molecular Design into Product and Process Design.

[25] Jennifer Ferreira, James Noble, and Robert Biddle. Agile development iterations and ui design. In *Agile 2007 (AGILE 2007)*, pages 50–58, 2007.

[26] Talita Stockle, editor. *UX Design Process.* Smashing Magazine, Freiburg, Germany, 2013. For information about this resource please contact CLoK@uclan.ac.uk.

[27] Matthew J. Hamm. *Wireframing Essentials.* Packt Publishing, Limited, 2014. ProQuest Ebook Central.

[28] Bradley Camburn, Vimal Viswanathan, Julie Linsey, David Anderson, Daniel Jensen, Richard Crawford, Kevin Otto, and Kristin Wood. Design prototyping methods: state of the art in strategies, techniques, and guidelines. *Design Science*, 3:e13, 2017.

[29] Michael Schrage. The culture(s) of prototyping. *Design Management Journal (Former Series)*, 4(1):55–65, 1993.

[30] Zulfiandri, Silma Novshienza Putri, and Aang Subiyakto. Evaluating user interface of a transport application using usability evaluation methods. In *2021 9th International Conference on Cyber and IT Service Management (CITSM)*, pages 1–7, 2021.

[31] Adrian Fernandez, Emilio Insfran, and Silvia Abrahão. Usability evaluation methods for the web: A systematic mapping study. *Information and Software Technology*, 53(8):789–817, 2011. Advances in functional size measurement and effort estimation - Extended best papers.

[32] Jakob. Nielsen and Rolf. Molich. Heuristic evaluation of user interfaces. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 249–256, 1990.

[33] Braulio Murillo, Jose Pow Sang, and Freddy Paz. Heuristic evaluation and usability testing as complementary methods: A case study. Lima, Peru, 2023. Pontificia Universidad Católica del Perú. Braulio Murillo also affiliated with DHL Express Perú, Callao, Peru.

[34] Mureen Allen, Leanne M. Currie, Suzanne Bakken, Vimla L. Patel, and James J. Cimino. Heuristic evaluation of paper-based web pages: A simplified inspection usability methodology. *Journal of Biomedical Informatics*, 39(4):412–423, 2006.

[35] Carol M. Barnum. *Conducting a Usability Test*, pages 1–2. Elsevier, 2011.

[36] Jorge Gabriel Siqueira and Melise M. V. Paula. Ipead a/b test execution framework. *Proceedings of the XIV Brazilian Symposium on Information Systems*, 2018.

[37] E Wang and B Caldwell. An empirical study of usability testing: Heuristic evaluation vs. user testing. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 46, pages 774–778, 2002.

[38] J. R. McKee. Maintenance as a function of design. In *Proceedings of the AFIPS National Computer Conference*, pages 187–193, Las Vegas, 1980.

[39] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*, pages 235–241, 2002.

[40] R. Gunning. *The Technique of Clear Writing.* McGraw-Hill, 1952.

[41] Raymond Buse and Westley Weimer. A metric for software readability. In *ISSTA'08: Proceedings of the 2008 International Symposium on Software Testing and Analysis 2008*, pages 121–130. ACM, 2008.

[42] Raymond P L Buse and Westley R Weimer. Learning a metric for code readability. *IEEE transactions on software engineering*, 36(4):546–558, 2010.

[43] Brad A. Myers and Jeffrey Stylos. Improving api usability. *Communications of the ACM*, 59(6):62 – 69, 2016. Cited by: 123; All Open Access, Green Open Access.

[44] Brad A. Myers and Jeffrey Stylos. The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 105–112, New York, 2008. ACM Press. Atlanta, GA, Sept. 23–27.

[45] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. Methods towards api usability: A structural analysis of usability problem categories. In *Proceedings of the 4th International Conference on Human-Centered Software Engineering*, pages 164–180, Toulouse, France, 2012. Springer.

[46] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, nov 1993.

[47] N Bjørndal, Antonio Bucchiarone, Manuel Mazzara, Nicola Dragoni, and Schahram Dustdar. Migration from monolith to microservices : Benchmarking a case study. 03 2020.

[48] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

[49] Alan Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.

[50] Ruoyu Su and Xiaozhou Li. Modular monolith: Is this the trend in software architecture?, 2024. Copyright - © 2024. This work is published under http://creativecommons.org/licenses/by-nc-sa/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2024-01-24.

[51] Nuno Gonçalves, Diogo Faustino, António Rito Silva, and Manuel Portela. Monolith modularization towards microservices: Refactoring and performance trade-offs. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, pages 1–8, 2021.

[52] Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 318–325, 2016.

[53] Michael Bostock. D3.js - data-driven documents. `https://d3js.org/`, 2023. Version 7.5.0.

[54] Mike Bostock. Streamgraph transitions. `https://observablehq.com/@d3/streamgraph-transitions?intent=fork`, August 2023. Accessed: 28-04-2024.

[55] Uber Technologies Inc. React-vis: A composable visualization system built on top of react and d3. `https://uber.github.io/react-vis/`, 2023. Accessed on [02/03/2024].

[56] Uber Technologies Inc. Company overview. `https://www.uber.com`, 2024. Accessed on [02/03/2024].

[57] React-chartjs-2. `https://react-chartjs-2.js.org/`. Accessed: 2024-04-28.

[58] Chart.js. `https://www.chartjs.org/`. Accessed: 2024-04-28.

6 Appendix

Included below is the project proposal for this literature review: