Temperature Stability in Consumer-Grade Semi-Automatic Espresso Machines

A dissertation submitted in partial fulfilment of the requirements for the MSc in Advanced Computing Technologies

by Luke Channings

Department of Computer Science and Information Systems
Birkbeck, University of London
September 2023

Abstract

Consumer-grade semi-automatic espresso machines are affordable appliances for making espresso at home. They allow a home barista to brew espresso, steam milk, and produce hot water. Producing a consistent and delicious tasting espresso at home is not easy and takes practice. Developing a profile for a perfect shot of espresso, known as "dialing in" the shot, requires an iterative process of weighing beans, grinding them to a specific consistency, evenly distributing the ground coffee in the basket, compressing the grinds evenly and finally pumping hot water through the grinds. Each part of this process needs to be optimised by the barista, and a more detailed and specific set of criteria will produce consistently delicious espresso.

Extraction temperature is the temperature of the hot water that is pumped through the ground coffee, and it is an important factor in producing a well balanced espresso with consistent results. An extraction temperature in the range of 90-100 ℃ is desirable. In this study, I use a machine-learning approach to measure, quantify, predict, and implement temperature control in a common home espreso machine. The increased control, when achieved, will allow the barista better information about the characteristics of their espresso-brewing process, but also allow them to optimize their process going forward.

Table of Contents

List of Figures

1. Introduction

Coffee, or *cafè*, is a beverage that has been consumed by humans since the 15th century [2, p. 48]. Legend says that coffee was discovered by a young 9th-century goatherd named Kaldi [3, p. 121], who noticed his goats acting strangely after eating the red "olives" of an evergreen shrub. Kaldi ate the berries himself out of curiosity, and felt energised and mildly intoxicated. Some tellings of the myth mention a Sufi monk passerby who inquired about the boy's behaviour, the boy confessed and the monk took some berries for himself to help him stay alert during his evening devotions [4]. The plant, now known as the *Coffea* genus, was cultivated in Ethiopia and Yemen where it was called *qahwa* - a word that at the time referred to wine. This caused coffee to become known as "The Wine of Islam" [5]. The modern European words for coffee are derived from this same root word — *qahwa*.

Traditional methods of preparing coffee include grinding roasted beans with a mortar and pestle until they are as fine as flour, and then placing the ground coffee in the bottom of a pot (*ibrik / cezve*), which is then heated over a fire. Later, hot sand became a popular heating method [6] since the heat is more even than a direct flame. Modern filter coffee can be produced with devices like a Chemex or V60, where ground coffee is placed into a conical paper filter and suspended above a carafe. Hot water is poured into the cone until it is full, the coffee dissolves into the boiling water and drains into the carafe below. Common drip coffee machines work similarly, but the process is automated and less precise.

Cafè espresso is a method of brewing small amounts of strong coffee very quickly. These single servings, known as shots, can be brewed in less than a minute. The best English translation for the word 'espresso' is debated, but based on marketing and etymology it can be taken to mean coffes that is brewed quickly, and on-demand [7] (as in *piatto espresso*), as opposed to coffee that has been prepared beforehand like filter coffee in an American diner. The exact definition of what espresso is has changed over time as methods of producing it have evolved. The patent for the first espresso machine was granted to Italian inventor Angelo Moriondo on the 16th of May 1884 [8]. The machine had a large vertical boiler, a gas burner located at the base, and a chimney running up through the centre of the boiler. The boiler would be half filled with water and heated. Steam pressure from the boiling water could be released through a valve located at the top of the boiler. A separate brew water tank, which is connected to the boiler via a tube controller by a tap, takes water from the boiler and regulates its release into the coffee grounds via a second tap.

The ground coffee is held in a filter basket and slotted into a dedicated space at the bottom of the brew water tank. When pressure in the boiler is sufficient, the barista — a word that means barkeeper in Italian, but here refers to the person operating the espresso machine — would turn the tap to fill up the brew water tank, and when the pressure had equalised and water had filled the tank, they could then turn the tap to brew the coffee. This machine used 500-600 grams of ground coffee and produce 50 servings at a time [9]. The temperature of the heated water in this machine would have been $\approx$100 ℃ since it used steam pressure to push the water through the coffee. The water that remained liquid must be close to 100 ℃, although some heat would have been lost as the water travelled to the coffee.

In 1938, Achille Gaggia — another Italian inventor famous within the espresso community — adapted Moriondo's multi-serving machine design into a single shot design. He patented the portafilter (a portmanteau of "portable" and "filter"), a removable device made up of a handle, a detachable filter basket where the ground coffee is placed, and a spout. Portafilters come in both single and double-spout varieties, allowing the barista to produce more than one shot at a time when necessary. There are also "naked portafilters", which do not have a spout but instead have an open bottom through which the brewed espresso can drain omnidirectionally.

Moriondo's brew tank is replaced by the grouphead in Gaggia's design, a block that the portafilter locks to. Water flows directly from the boiler and through the grouphead; the flow of water was originally controlled by a manual value, and later an electrical solenoid valve. The water from the grouphead is dispersed over the portafilter basket using a dispersion screen, also known as the shower screen.

In 1947, following the success of his first machine, Gaggia recieved the first patent for a piston-driven extraction mechanism [10]. This design uses a lever to compress a spring, which when released forces a piston through a water chamber and into the ground coffee. The modern colloquial term for producing a shot of espresso is "pulling a shot", in reference to the piston machine's lever [11]. This innovation allowed the brewing water to be below 100 degrees celsius, since steam pressure was no longer needed. The spring usually produced higher brew pressures (9-10 bars, or 116-145psi) than the steam method (which can vary depending on the pressure valve, but is reported as being between 1.5-2 bars, or 7-22psi [12]), which means that the coffee must be ground finer in order to pull a shot in the same time. As the coffee is ground more finely, the increased surface area allows more flavour (measured in Total Dissolved Solids, TDS) to be extracted.

In 1961 the Faema E61 [13] was released, which was the first espresso machine to use an electrical pump. The design for the E61 grouphead and portafilter became a standard in espresso machines across the industry, meaning portafilters and shower screens can be used interchangeably between a variety of espresso machines today.

In 1998, the Italian Espresso National Institute was founded to safeguard traditional Italian espresso. It published a technical definition for "Certified Italian Espresso" that same year (Table 1). The parameters published by the IENI describe a "recipe" for Traditional Italian Espresso as it is still being served in Italy today. This traditional espresso is typically made with darkly roasted beans which are more porous and brittle, making them easier to extract [2, p. 105]. This style of espresso is typically a blend of espresso beans, for example 80% *Coffea arabica* and 20% *Coffea canephora* is a common blend.

Necessary portion of ground coffee	7 g ± 0.5
Exit temperature of water from the unit	88 ℃ ± 2 ℃
Temperature of the drink in the cup	67 ℃ ± 3 ℃
Entry water pressure	9 bar ± 1
Percolation time	25 seconds ± 5 seconds
Viscosity at 45℃	> 1.5 mPa s
Total fat	> 2 mg/ml
Caffeine	< 100 mg/cup
Millilitres in the cup (including froth)	25 ml ± 2.5

Table 1: Certified Italian Espresso parameters [1], scaled for 1 serving

However, as coffee culture expands into a global phenomena, experimentation and individuality in brewing has gained popularity in the last few decades. Strict adherence to these standards is not always expected by coffee experts and hobby baristas, nor is it always preferred by untrained drinkers. *Coffea canephora*, used in traditional espresso blends, is cheaper to produce but is said to have a "burnt rubber" taste in the cup. [2, p. 12]. Known popularly as Robusta coffee, *canephora* beans are commonly associated with cheap and low quality commodity coffee.

Commodity coffee is used to refer to coffee that is produced and consumed as a commodity like sugar or flour, wherein scale and efficiency are more important than the quality or origin of the beans. Often commodity beans are a blend from many producers, rendering the beans impossible to trace back to their grower. *Speciality coffee* [14] juxtaposes commodity coffee, emphasising quality and traceability. Speciality coffee has more in common with the fine wine market than the sugar or flour markets: low-volume and single origin coffees are preffered by specialty coffeehouses, and the coffee roasters therefore tend to have a direct relationship with small, often family-owned coffee producers. Today, coffee enthiusiasts refer to three waves of coffee consumption. The first and second waves saw the popularisation and commoditisation of coffee throughout Europe: the first wave introduced coffee to Europe in the late Medieval period, while the second wave began with the invention of espresso at the turn of the 20th century. In the last few decades, a third wave of coffee has seen an explosion in choice of high quality coffees as well as the way these coffees are prepared. The espresso community has adopted lighter roast coffees, not just the dark roasts that are a hallmark of Traditional Italian Espresso.

Extraction temperature (temperature of the water as it exits the unit and contacts the coffee) has a large impact on the quality of the final cup. Increasing temperature increases the extraction of soluble compounds in the coffee, of which only ≈ 30% are water soluble [15]. Dark roasts are normally extracted between 85-95 ℃. Lighter roasted beans are harder to extract, and temperatures of 90-95 ℃ are recommended [16]. The ability to set the extraction temperature is therefore valuable. Temperature consistency is also important, and whilst controlled temperature gradients may be desirable, uncontrolled temperature gradients make shots impossible to reproduce and risk the temperature falling out of the narrowly-defined acceptable range.

More soluble compounds are extracted at higher temperatures [17]. Extracting all 30% of solubles into the beverage would extract bad-tasting compounds that have been described as "bitter" and "papery" [15]: this is known as *over extraction*. Extracting fewer soluble compounds, which can be done by pulling a shot for a short time or grinding too coarsely, produces a sour-tasting coffee. This is called *under extraction*.

Unlike dose (the amount of coffee used), grind size, and extraction time, the extraction temperature is not under the direct control of the barista. This means that the barista will not be able to consistently pull a shot of the same coffee. These changes in temperature can lead to under-extraction or over-extraction of any particular shot of espresso, leading to an inconsistent taste despite use of the same beans, at the same grind size, and with the same machine and barista. The ideal temperature for the brew will depend on the specific coffee's roast level, with darker roast beans needing higher temperatures and medium roast beans needing lower temperatures. When the extraction temperature is not known to the barista, they will not be able to improve their extraction. As third-wave expert and hobbyist baristas strive for better results and more interesting taste profiles, increased control over the brewing parameters are highly sought-after.

This project will focus on a consumer-grade espresso machine: the Rancilio *Silvia*. This machine has a single boiler which has the dual purpose of heating water for brewing espresso and producing steam, which when expelled through a steam wand can be used to produce steamed milk. Espresso combined with varying quantities and textures of steamed milk can be used to make popular drinks such as the latte, cappuccino, flat white, and others. This machine is a mid-range option when compared to all coffee brewing systems, but is considered entry-level for home espresso brewing. It is a popular and accessible choice in Europe and North America, and the results of this study are therefore generalizable to a large consumer share.

The Italian Espresso National Institute defines one more parameter in its definition of espresso: brew pressure [18]. As with all other aspects of the brewing process, pressure profiling and flow profiling are a current topic of research amongst coffee enthusiasts [19]. Although water pressure will affect the espresso made by any machine, most entry-level and mid-range home espresso machines, including the Rancilio Silvia, do not have a mechanism for changing the pressure. This is a feature of higher end commercial and "prosumer" espresso machines. Unlike pressure profiling, extraction temperature stability can be improved without expensive hardware.

The boiler temperature and consequently the extraction temperature for the Silvia is controlled by a temperature-sensitive actuator (TSA), which is either open or closed (heating or not heating the boiler) when the boiler temperature is below or above 100 ℃ respectively. The temperature sensor is fixed to the outside of the boiler and the heat generated within the heating element of the boiler takes some time to travel from the element to the TSA. Because of this *temperature lag* between the sensor and the heating element, the boiler will continue to heat up after the TSA has closed. The consequence of this is that the boiler temperature can vary by ≈10 ℃ (Figure 1) from one shot to the next, depending how long ago the boiler was turned off. I call the machine's default method of controlling temperature "threshold control", since its behaviour is determined by the boiler temperature crossing a threshold.

Most espresso machines need to be pre-heated before coffee can be brewed because some of the heat from the boiler will escape into the machine's casing, causing the brew water to be cooled as it travels from the boiler to the coffee. A machine that has been up to temperature for a period will heat up the machine's chassis, which reduces the cooling effect. Machines vary in the amount of time required for them to pre-heat, and the exact temperature difference between a cold machine and a pre-heated machine is not known.

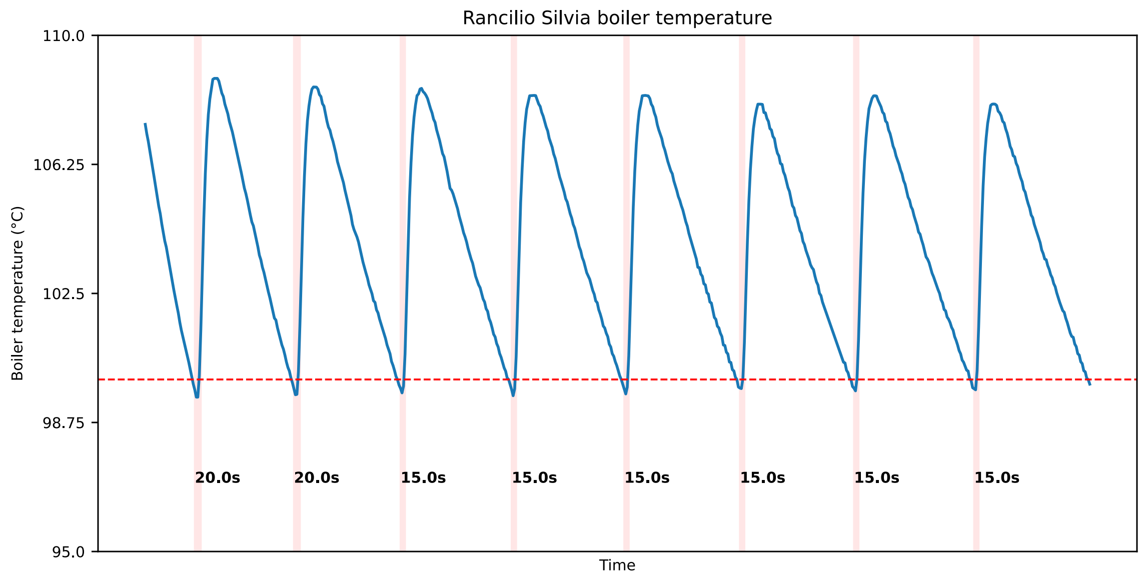


Figure 1: 100 ℃ Threshold Control Method

In this project, I plan to improve results in the problem of inconsistent and opaque extraction temperatures when brewing with the Rancilio Silvia. This will include:

1. Hardware modifications to enable programmatic control over the machine's power and heating element, and temperature sensors on the boiler and grouphead.
2. Software to model the machine's state and provide an interface (API) for interacting with it, as well as a mechanism to store and query past machine states constituting a rich dataset of the machine's behaviour.
3. An end-user interface for viewing real-time temperature data, recording when a shot is being pulled (further contributing to a dataset), and controlling the machine's behaviour.
4. Data analysis to learn about the behaviour of the machine through gathered data
5. The production of predictive models to enable a new predictive temperature control method

2. Existing solutions

A common method of mitigating this problem is **temperature surfing**. The barista will intentionally discharge hot water through the grouphead until the heating indicator the machine illuminates, and then waits 10 seconds after the indicator turns off before pulling their shot. By following this procedure the boiler will be at approximately the same point in the heating cycle each time the barista pulls a shot. This results in a more consistent temperature, but they won't be able to know or adjust the extraction temperature, only that it is approximately the same each time a shot is pulled.

Another solution that relies on similar hardware changes to the ones I propose is the addition of a PID controller. Popular projects such as PID Silvia [20], Auber PID [21], and meCoffee [22] provide hardware modification kits that keep the boiler temperature stable using a PID controller.

I do not believe that a PID solution will perform well considering the temperature lag I have described, as even a well tuned PID controller will not accurately control a system where there is significant delay in measurements [23]. These projects may use a lower latency temperature sensor, or perhaps mount a sensor closer to the boiler's heating element, but this remains to be seen. I note specifically the graph shown in the meCoffee product page [24], which shows a temperature variance of 20 ℃ with the unmodified Silvia, which contradict my measurements (Figure 1) that show the variance is closer to 10 ℃. A PID controller is a control system that works best when the sensor lag is as close to zero as possible, and when there is a lag the PID controller will have a similar performance profile to the threshold control method.

My proposed solution uses a predictive modelling approach which may not have the same sensitivity to sensor lag that a PID controller will suffer from.

3. Hardware

3.1. Rancilio Silvia

The Rancilio Silvia has two model variations: *E* and *M*. The *E* model has an auto-off function that satisfies the EU's idle energy use regulations. The *M* model omits this auto-off function and is sold outside of European markets. Both models allow the barista to brew espresso, steam milk for espresso-based drinks, and to dispense hot water through the steam wand for americanos, tea, and other hot-water beverages. In this project, an *E* model is used.

The standard wiring of the Rancilio Silvia is shown in Figure 2, with its stock functions described in the following sections.

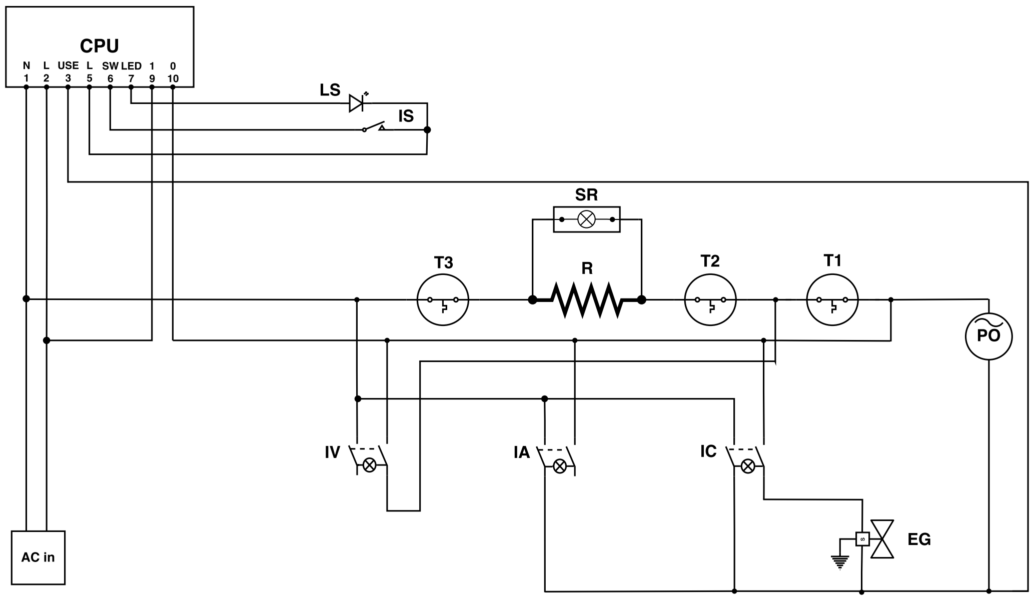


Figure 2: Rancilio Silvia E Electrical Layout

3.1.1. Power on

The machine is powered on when the spring-loaded power switch **IS** is toggled. This illuminates the power light **LS**, and the CPU starts a 30 minute auto-off timer, electrifying **CPU** pins 9 and 10. The wire coming in to **CPU** pin 3 will be electrified whenever the machine is in use, and it will reset the auto-off timer, thus preventing the machine from automatically powering off whilst being in use.

As soon as the machine is powered on, the boiler begins heating (see the element labelled **R**). While the element is heating, the heating light **SR** will be lit. The temperature-sensitive actuator (**TSA**) **T1** will open when the thermocouple detects a temperature greater than 100 ℃ - this will disable **R** and **SR**.

3.1.2. Brew

The barista grinds coffee into the filter basket attached to the portafilter, which is then locked to the grouphead after some additional preparation.

When the machine has been heating for a sufficient period of time and is ready to brew, the brew switch **IC** is toggled by the barista. The water pump **PO** pumps water from the water reservoir to the boiler. The solenoid valve **EG** opens, creating a path for water to pass from the boiler and into the grouphead.

The ground coffee in the filter basket provides resistance to the water which creates pressure as the pump pushes water into the boiler. The pressure forces the water through the coffee, producing espresso that falls into a cup placed underneath the portafilter.

3.1.3. Hot water and Steam

The **IA** switch is for hot water, and it effectively does the same thing as the brew switch except the solenoid valve **EG** is not opened. Hot water from the boiler is pressurised and when the steam wand valve is opened by the barista (by turning the steam knob), hot water from the boiler is pumped through the steam wand.

The **IV** switch is for steaming milk. The switch bypasses **T1**, **EG**, and **PO**, causing the **T2** TSA, rated to 140 ℃, to control the boiler temperature. This creates pressurised steam in the boiler, which can be exhausted through the steam wand by the barista in the same manner as they dispensed hot water. Pressurised steam is used to aerate and heat milk, which creates a silky-textured hot milk that blends with the espresso.

The third TSA — **T3** — is a safety shut-off rated for 165 ℃. In case the **T1** and **T2** are bypassed or broken **T3** will prevent the boiler from exceeding ≈180 ℃.

3.2. Raspberry Pi Zero

I use a Raspberry Pi Zero as the primary platform for hosting the software and integrating hardware modifications. The Pi Zero (Figure 3) is a small single-board-computer (SBC) that I mount inside the casing of the Rancilio Silvia. The device includes a General-purpose Input/Output interface (**GPIO**) that I will use to interface with temperature sensors and relays. It is powered independently of the Silvia, receiving power via a USB cable that is routed underneath the Silvia's chassis. The Pi Zero supports WiFi: however, due to the potential signal problems of being housed inside of a metal case and the response time requirements of the software, I have opted to connect the Pi Zero to the network via an ethernet cable. This is achieved by using a micro-USB 100BASE-T ethernet adaptor, as the Pi Zero does not include an ethernet device onboard.

The Pi Zero is a low performance device, with a 1GHz single-core 32-bit ARM System-on-Chip (SoC), 512MB of RAM, and microSD storage. A faster SBC (the Pi Zero 2) was released in 2021. Due to the electronics supply chain problems and ongoing shortages caused by the COVID-19 pandemic [25], I have been unable to obtain this device for my project. When it is available, I plan to upgrade as the 64-bit architecture is better supported and my application would benefit from multi-threading.

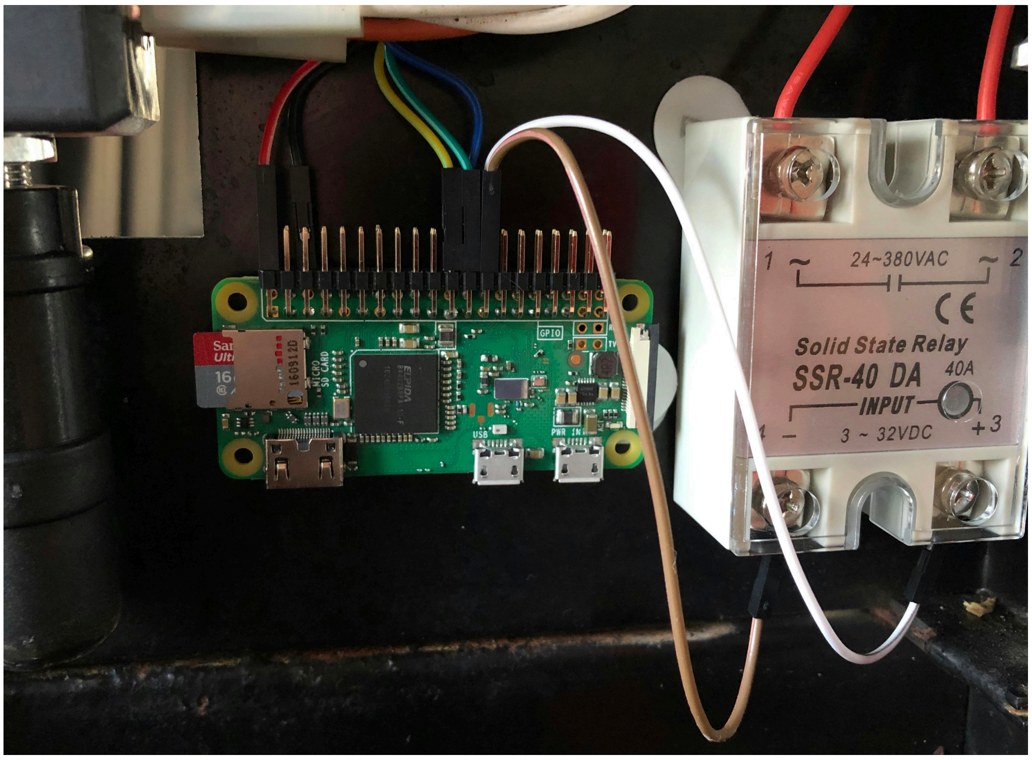


Figure 3: Pi Zero mounted inside the housing of a Rancilio Silvia

3.3. Modifications to the Rancilio Silvia

Several hardware changes are required to perform this project. In order to build the software platform for controlling the brew session, I need to have hooks into the machine that allow the programmatic retrieval of important features, such as the machine's power-on state, the boiler temperature, etc.

3.3.1. Power Relay

I need control over the machine's power state for these reasons:

1. If the machine is always assumed to be powered on, recordings of the machine's state could be difficult or impossible to model. For example, if we are measuring how much the temperature increases with various heat levels but the machine is powered off, the measurements would be invalidated because although our software is turning the boiler on nothing is happening because the machine is powered off.
2. A feature of the software will be to programmatically run experiments like the heat measurement example above. Having no direct control of power status makes this impossible.

It is possible to control the power state of the *M* model by using smart power plug, but the auto-off mechanism used by the *E* model makes this impossible, since the power state of the machine is managed by the CPU component and requires manual actuation of the **IS** power switch. To solve this problem I use the Shelly 1 [26] — a small and cheap ESP8266-based WiFi relay switch. The ESP8266 is a chipset that is supported by the ESPHome project, which is a customisable firmware for integrating ESP chipset devices with IoT infrastructure.

The configuration and instructions for flashing the Shelly switch can be found in my source code under `config/relay_switch`.

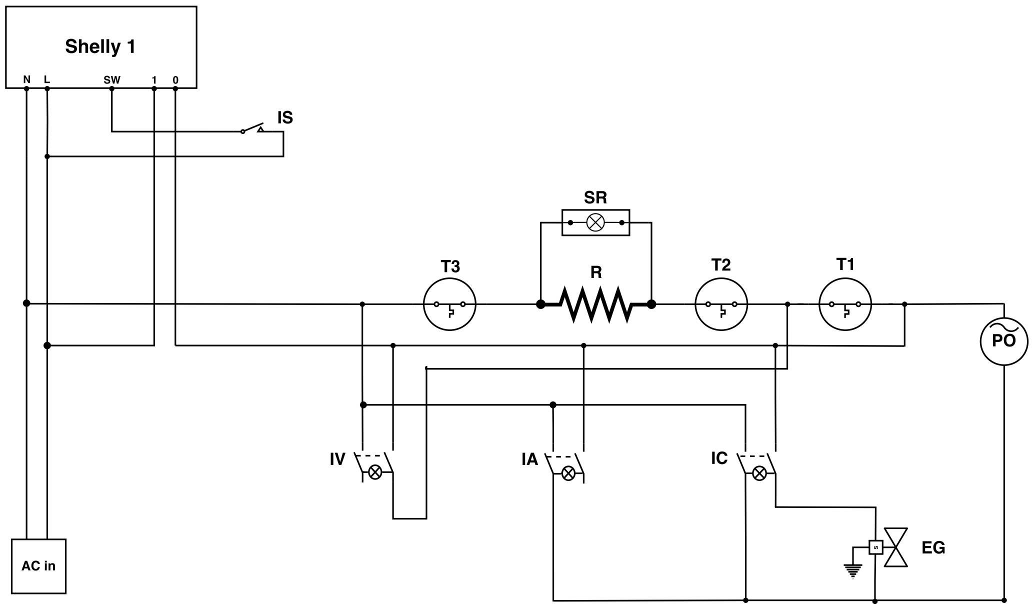


Figure 4: Silvia & Shelly 1 Electrical Layout

The layout is modified as follows (Figure 4):

- Removal of the CPU component and replacement by the Shelly 1
- Removal of the cable going to CPU pin 3, which is used to sense whether the machine is being used and delay the auto-off function
- Disconnection of the power LED (denoted **LS**) — a 3.3V DC LED will be wired to the Raspberry Pi instead.

The relay will connect to an MQTT broker to integrate with the rest of the software, this is described further in the Software Architecture section.

3.3.2. Boiler element control

The software is required to control the boiler's heating element, which will enable the implementation of novel control methods. There must be hardware to support this, which requires the modification of the machine's electrical layout.

Boiler control requires two things:

1. A power relay that can be integrated with an embedded computer system that can be used to toggle the boiler's heating element, and
2. A temperature sensor to monitor the boiler's temperature

As discussed, the **T1** thermocouple controls the temperature of the boiler during normal operation. It is a temperature-sensitive actuator (Figure 5). The underside is affixed to the boiler, and the left and right pins will complete a circuit to the boiler's heating element when the temperature is below the actuation threshold, thus heating the water.

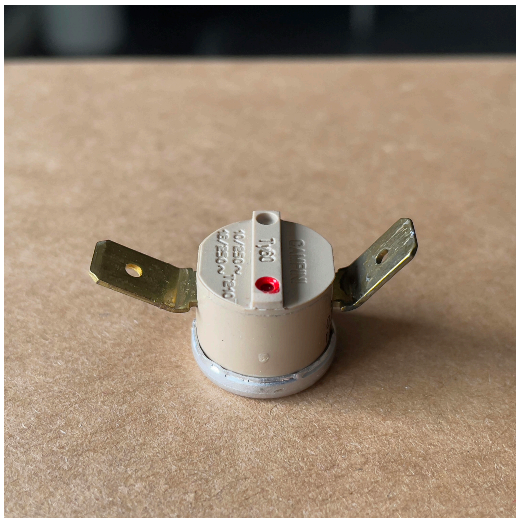


Figure 5: A temperature-sensitive actuator from the Rancilio Silvia

I modify the machine as follows:

- Remove the **T1** TSA
- Place a Type-K thermocouple - insulated by a non-conductive silicone pad - where the **T1** TSA was previously fixed to the boiler.
- Wire the pins previously connected to the **T1** TSA to a Solid-State Relay (SSR)

The SSR I have chosen to use is the Fotek SSR-40 DA. The datasheet [27] states that it responds within 8.3ms and is zero-crossing, meaning that the power change will only be made when the AC power waveform goes to 0.

I attach the Type-K thermocouple to a MAX31855 [28] thermocouple amplifier, which is wired to the Pi Zero's GPIO interface.

Other thermocouples such as a Resistance Temperature Detector (RTD) with a PT100 sensor are also possible. The Type K + MAX31855 combination was chosen due to its low cost and ease of integration. The Type K thermocouple is an inexpensive thermocouple that can sense temperatures between -200 ℃ to +1350 ℃. Its temperature response curve is nonlinear, but only at low (< 0 ℃) temperatures, so temperature gradients are not a concern in this application.

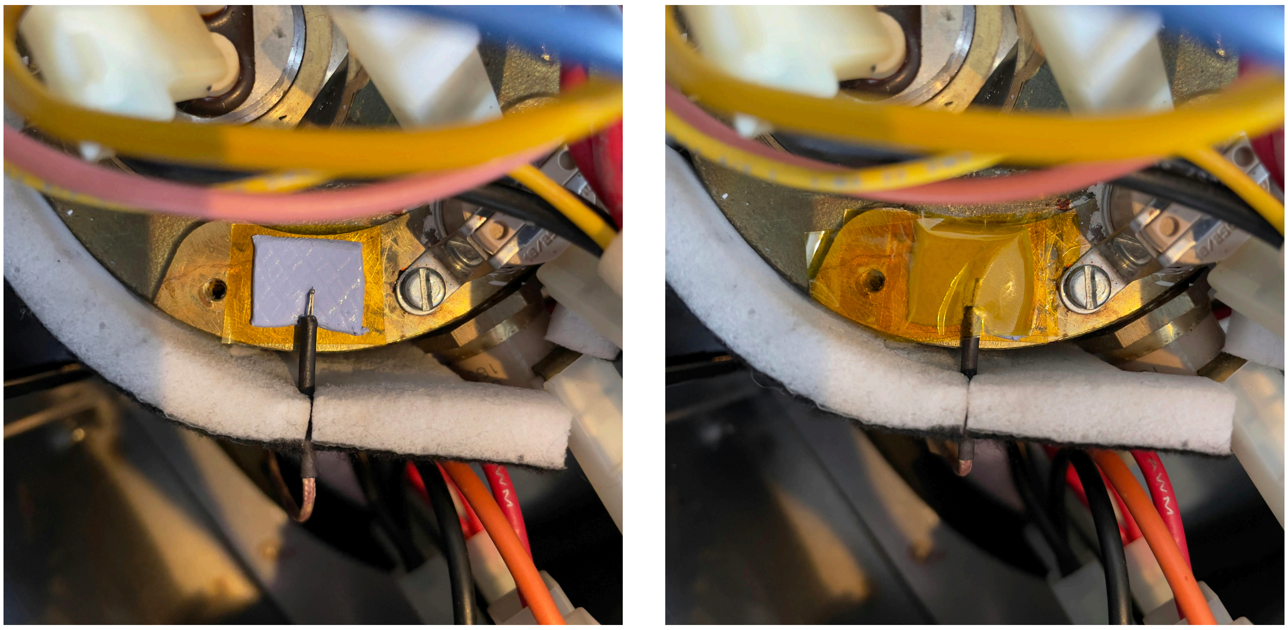


Figure 6: Thermocouple against a non-conductive thermal pad (left), and secured in place (right)

The thermocouple works by joining two wires end-to-end, one made from chromel and the other from alumel. The difference in resistance between the two metals is used to determine the temperature at the junction. This requires running a current through the wire, and care must be taken to ensure the thermocouple probe does not contact other conductive surfaces. I insulate the probe from the boiler with a temperature conductive silicone pad.

3.3.3. Grouphead thermocouple

The grouphead heats up with the boiler and can be used to determine the degree to which the machine has pre-heated. Placing a probe on the grouphead itself is ideal because it is the primary component that the boiler water will lose heat to, as well as being the water's last point of contact with the machine before it floods the coffee grounds.

I modify the machine further: the machine front cover is disassembled, exposing the grouphead block. I place a non-conductive thermal pad on the metal and then place the thermocouple on top of that (Figure 7). I secure the thermocouple in place with a high-temperature polyimide film adhesive. The Type K thermocouple is attached to a second MAX31855 adaptor, which is then wired to the Pi Zero's GPIO.

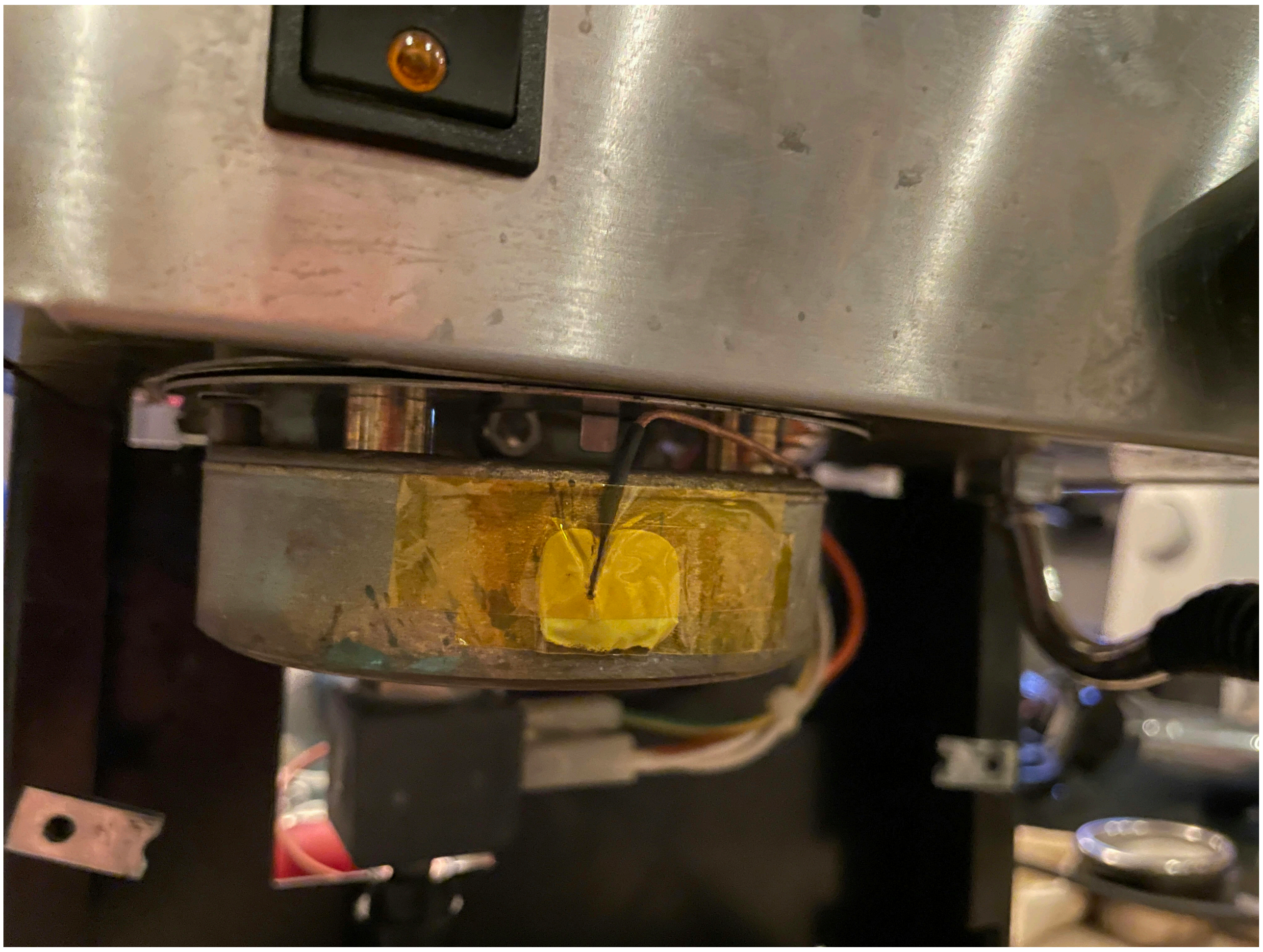


Figure 7: Type K thermocouple attached to the grouphead

3.3.4. Final wiring

The Pi Zero's SPI0 interface has 2 **Chip Selects**, meaning at most two MAX31855s can be wired to that interface. The third MAX31855 can be used to connect the thermofilter, and is wired to the SPI1 interface, but this will not be used. I will leave support in place in case it is needed later. The SSR is connected to GPIO 26. This concludes the hardware modifications that have been made to the machine. A diagram showing the pinout from the Pi Zero's GPIO to the SSR and MAX31855 thermocouple amplifiers is shown in Figure 8. The final electrical layout of the Rancilio Silvia after modifications is shown in Figure 9.

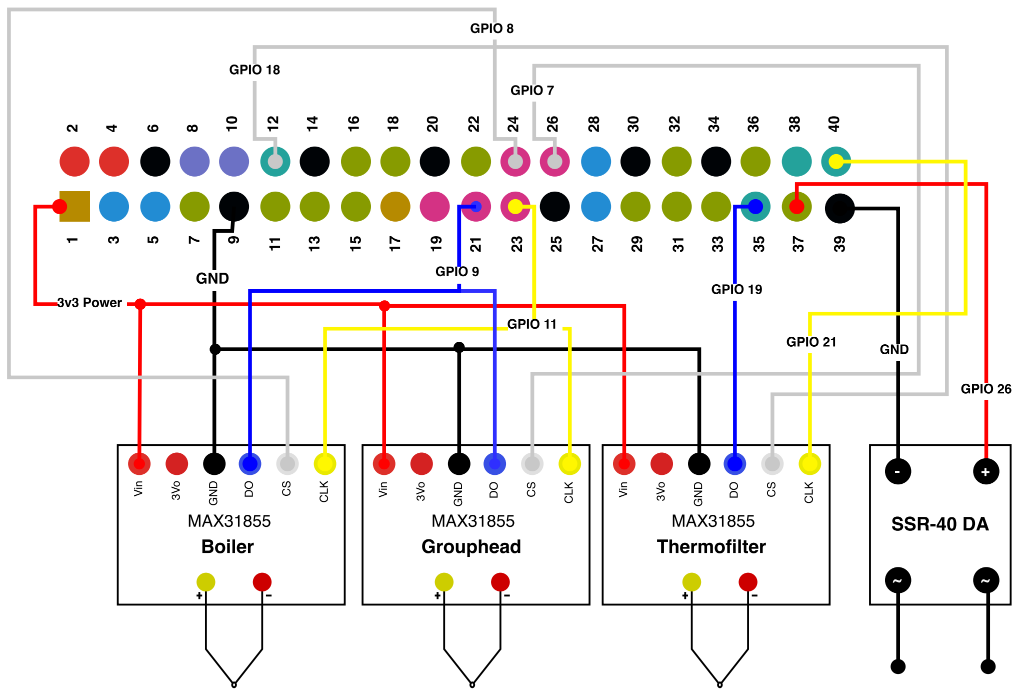


Figure 8: Pi GPIO, MAX31855, & SSR wiring diagram

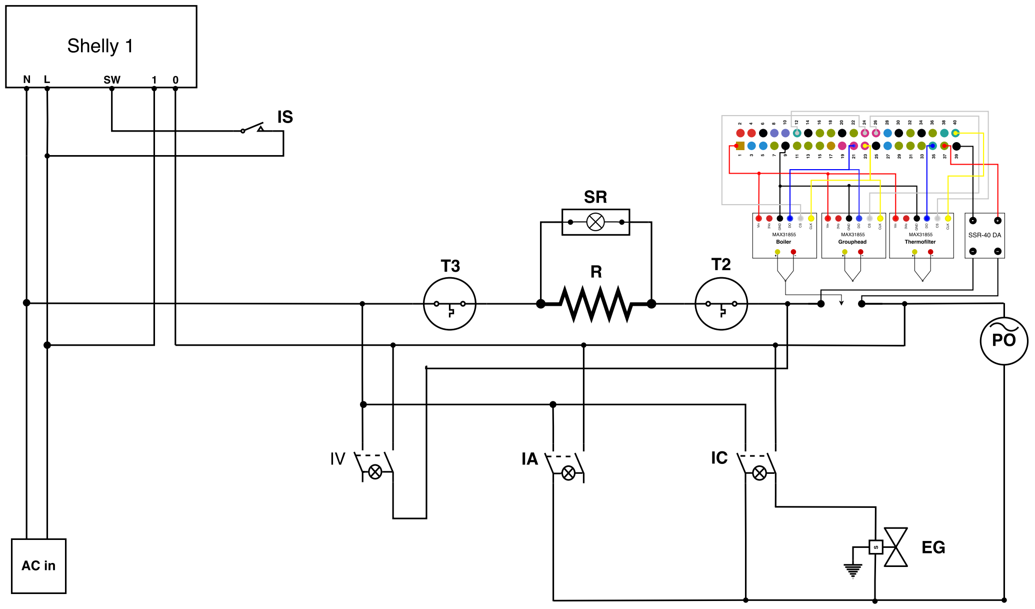


Figure 9: Complete wiring diagram including all modifications

3.4. Thermofilter

The primary variable that I want to measure is the temperature of the water as it exits the machine and permeates the coffee: the *extraction temperature*. Understanding the relationship between the grouphead, boiler, and extraction temperatures is therefore required.

To measure the extraction temperature directly, I built a DIY measurement device called a thermofilter (a portmanteau of 'thermocouple' and 'portafilter') following specifications from a popular barista forum [29]. A thermofilter is a temperature sensor that is mounted to a portafilter. It is possible to measure the temperature that the coffee basket will experience by locking the thermofilter to the grouphead and pulling a shot as normal. There are commercial devices available, such as the Scace 2 [30], but the cost is prohibitively high for this study.

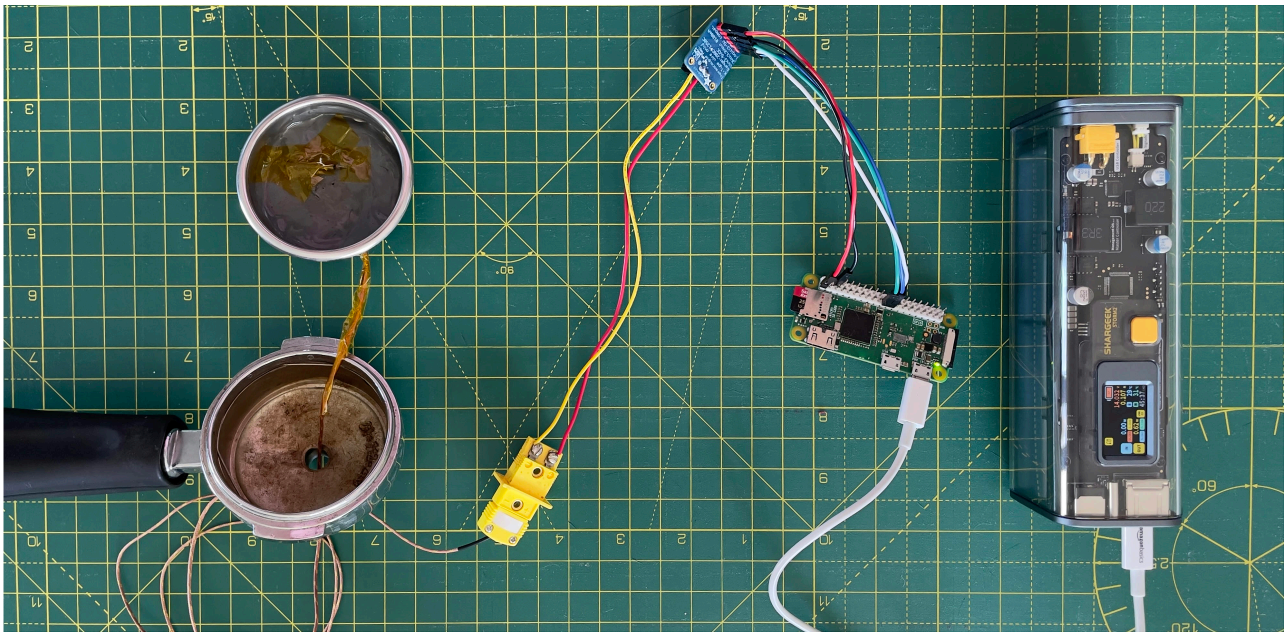


Figure 10: Thermofilter assembled with Pi Zero, MAX31855, & battery. 1/2 inch scale.

I construct the thermofilter using a blank basket which I drill two holes into: a small hole in the centre through which water can drain to the cup, and a larger second hole off-centre that the thermocouple can be fed through. Before feeding through, I wrap the thermocouple wire in Kapton tape, which is non-conducting and temperature resistant. This will protect the wire from the sharp edges of the metal hole.

When the thermocouple is in place, I add Kapton tape (a silicone-adhesive tape that can handle very high temperatures without melting) on the exterior centre hole. I then fill the basket with a shallow layer of epoxy resin —. Filling it too high will cause the epoxy to contact the grouphead's shower screen and cause water to run off the sides of the portafilter instead of into it. Before the epoxy dries I push an acupuncture needle through the centre hole in order to leave a passage for the water to drain through.

When the epoxy resin is dry, I remove the needle and tape the thermofilter to the surface of the resin to prevent it from moving during extraction (Figure 11).

In experimenting with the device, I found that there would sometimes be electrical interference from the machine when the thermofilter was connected to the Raspberry Pi. I have not thoroughly investigated the cause of this interference, however I have observed that it disappears when the thermofilter is isolated from the machine. I achieve this by using a second Pi Zero connected to a battery to gather measurements (Figure 10).

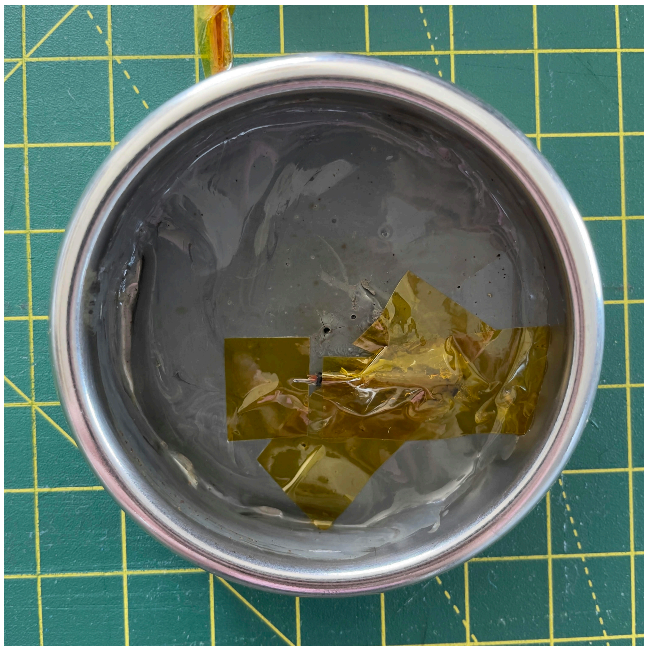


Figure 11: Thermofilter basket with thermofilter taped in place. 1/2 inch scale.

4. Software

Several software packages are required for the completion of this project. There are *n* distinct components, the *core*, which is written in Rust, The *user interface*, which is written in TypeScript, and the analysis and *modelling*, which is done in Python. I call the collection of software that supports this project *Gesha*, after a rare varietal of the *Coffea arabica* plant that has a fruity and tea-like flavour.

Now that the hardware platform is in place, I design and implement the software architecture. The first step is to define requirements for the software.

The core of the application will be run on the Pi Zero. It will be responsible for direct control of the espresso machine, including abstractions like executing temperature control methods and low-level tasks like reading temperatures from the MAX31855 modules.

The user interface (UI) is responsible for the presentation of the temperature data allowing the user to interact with the core application running on the Pi Zero. The UI can be run from anywhere, but must support the real-time presentation of temperature data and enable fine control over the machine's state. In my experimentation, I use a 2018 Apple iPad.

A third piece of software is used for extracting measurement data from the core application and processing it into a form that can be used for analysis or for producing predictive models.

Additionally, the thermofilter software must be integrated such that the temperature reading can be shown in real-time in the UI and the temperature data can be stored and combined in the processing phase.

4.1. Source Code

The source code can be found on GitHub at `Birkbeck/msc-project-source-code-files-22-23-LukeChannings`.

The project contains the following:

- `src/` - the main application is written in Rust, and dependencies are managed with Cargo.
- `ui/` and `src-tauri/` - the UI source code is written in TypeScript, and dependencies are managed with NPM.
- `models/` - projects for modelling some aspect of the machine's behaviour, written in Python, with dependencies managed by Poetry.

- `docs/` - the source files for my project proposal and dissertation, written in Typst.
- `data/` - contains the SQLite database that model datasets use.

The project is managed with a Justfile: run `just --list` for a list of recipes.

Note that any *code reference* comments will include links to the file in the GitHub repository.

4.2. Architecture

The application has components that are distributed across several devices: a Pi Zero, the Shelly relay switch, a computer or tablet running the UI, another Pi transmitting thermofilter readings, etc. Due to the distributed nature of the system, I have chosen to use the MQTT [31] protocol. MQTT is a topic-based publisher-subscriber asynchronous messaging protocol. An MQTT service is called a broker. I have chosen the Eclipse Mosquitto [32] broker because it is high performance and lightweight, and as such it can run comfortably on the Pi's performance-constrained hardware.

MQTT was chosen for the following key reasons:

1. The application requires real-time streaming of sensor data
2. The application must respond in real time to state changes in the relay switch
3. MQTT is a commonly used and well-supported protocol in IoT devices.
4. Using MQTT allows simple integration into automation tools such as Node-RED [33] and Home Assistant [34].

It is possible to fulfil these requirements using Web Platform techniques such as Webhooks [35] and Server Sent Events [36]. However, MQTT is simpler and provides better ergonomics for asynchronous APIs. Additionally, debugging or observing the system is straightforward with MQTT, since a client can connect and observe any messages.

Figure 12 shows an overview of the system components and how they interact. The Shelly 1 relay connects to the MQTT broker and publishes its state. It also provides a command topic to allow the relay to be toggled programmatically.

MQTT messages marked with the *retain* flag will be re-sent to connecting clients. When Gesha core connects to the MQTT broker, it will receive the current state of the power relay which will then be stored in the State Controller.

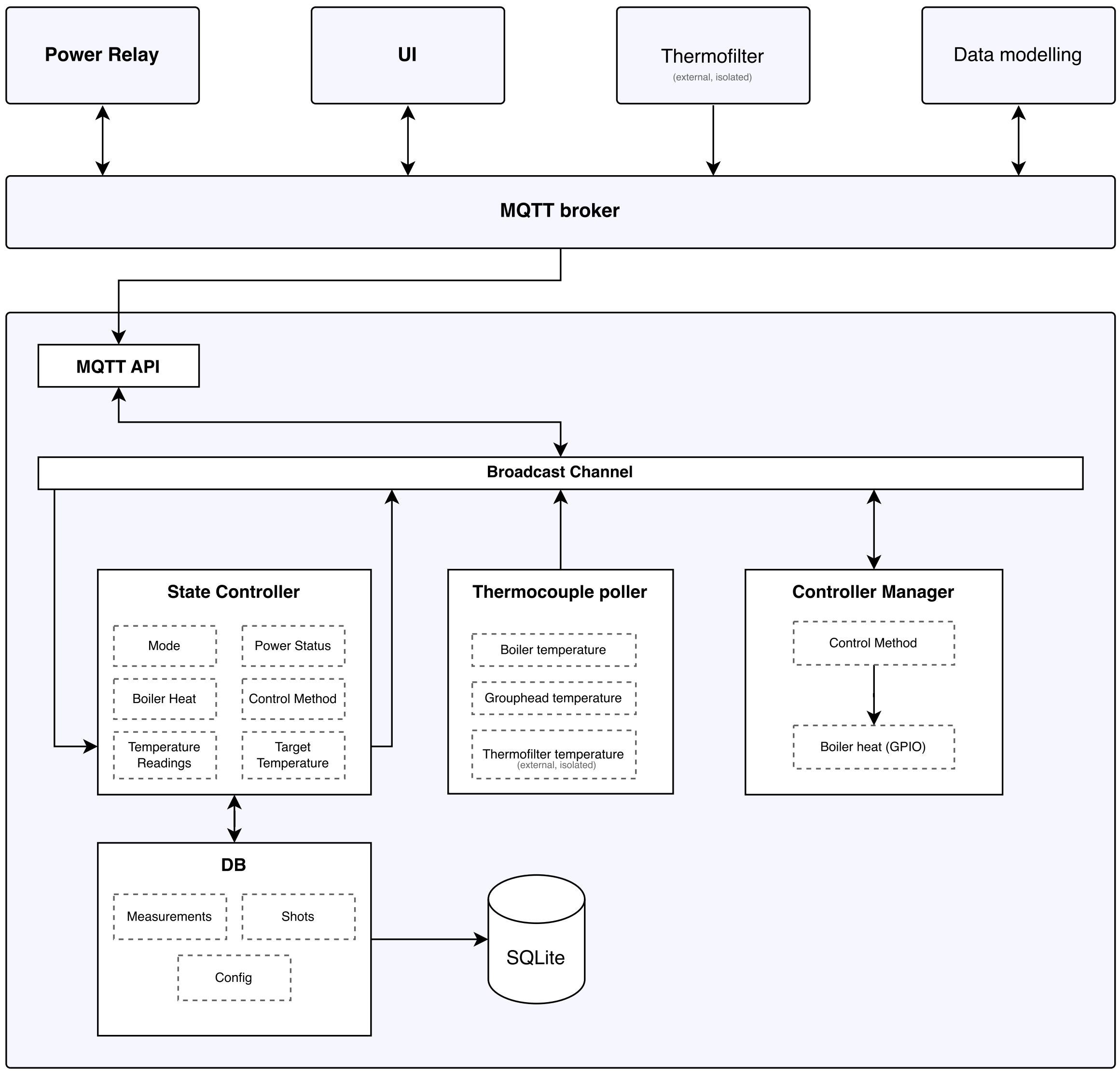


Figure 12: Gesha architecture overview

4.3. Core

The core application must be lightweight since it will run on performance-constrained hardware — the Pi Zero has a 1GHz CPU and 512MB of RAM. The storage device will be a micro-SD card, meaning read/write performance will be poor.

The application is written in Rust, which I have chosen because it satisfies the constraints I have in terms of performance, and high-level features like async/await without the overhead incurred by spawning and managing threads. Other languages such as Go and Python were considered, but Rust was chosen because of its memory management design (there is no garbage collection, and therefore the application does not pause) and high quality packages (crates).

The control flow of the application mirrors the wider pub-sub architecture. Modules within the application transmit an `Event` on the *Broadcast Channel*. Any events that are broadcast are explicitly handled by the State Controller, which may then publish MQTT events.

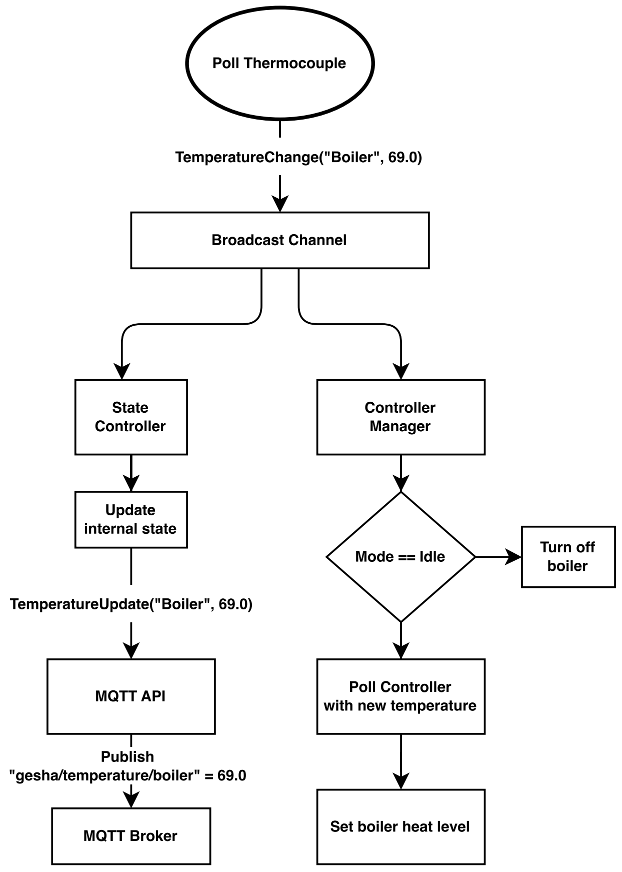


Figure 13: Example of a TemperatureChange propagating through the system

An example (Figure 13) is the *thermocouple poller*, which publishes `TemperatureChange` events to the Broadcast channel at a particular interval — when the State Controller receives a `TemperatureChange` event it updates its internal state and then publishes a `TemperatureUpdate` MQTT message, which updates the `gesha/temperature/{sensor}` topic. When the *Controller Manager* receives this event it samples the current control method instance (e.g. Threshold, PID, Predictive, etc.) with the new temperature, the result of which determines the new boiler heat level.

4.3.1. State Controller

The State Controller is a state machine that implements a `handle_event` function with the signature:

```
async fn handle_event(&mut self, event: Event) → Result<Vec<Event>>
```

Whenever an event happens in the application (with the exception of outgoing MQTT messages) the `state.handle_event` function is called. `handle_event` updates the application state and responds with further events (see Figure 13).

The machine can be in one of four modes: idle, active, brew, and steam.

The idle mode indicates that the Silvia is powered off, note that the Silvia's power state is controlled by the relay switch, and is independent of the Pi's power source. When the Silvia is powered off the software will still record the sensor temperatures, but it will not attempt to control the boiler.

The machine can be put into active mode by toggling the power switch on the machine, or by changing the mode to active through the software. When in active mode the machine is powered on and boiler control will be active.

The brew mode is identical to the active mode, but is used to mark measurements as being made whilst the machine was actively brewing. When a brew session ends with the user transitioning out of brew mode, a new show record is added to the shot table, which records the duration, start / end time, and average temperature during the shot.

The steam mode sets the temperature to 130 ℃ and measurements made whilst in steam mode have the steam column set to true. The barista could manually use the machine's steam switch to enter brew mode, but by using the software steam mode the recorded measurements are not invalidated. For example, if the software is in active mode but the machine is in steam mode the measurements will record a steep rise in temperature but the heat level recorded will be 0, this will make analysis harder.

	Idle	**Active**	**Brew**	**Steam**
Power	Off	On	On	On
Controller	Inactive	Customisable	Customisable	Over-ridden to Threshold
Thermocouple poll interval	1s	100ms	100ms	100ms
Target temperature	Inactive	Customisable	Customisable	Over-ridden to 130 ℃

Table 2: Software behaviour for each mode

Code reference: src/core/state.rs

4.3.2. Controller Manager

The Controller Manager is responsible for instantiating the configured Controller Method, which can be customised at any time.

A `Controller` must implement the trait:

```
pub trait Controller: Send + Sync {
    fn sample(&mut self, boiler_temp: f32, grouphead_temp: f32) → f32;
    fn update_target_temperature(&mut self, target_temp: f32);
}
```

This requires the controller to implement a `sample` function, which will take the current boiler and grouphead temperatures and return a number between `0` and `1` that indicates the new boiler heat level, where `0` turns the element off and `1` turns it on. The Controller Manager calls the current controller's `sample` function and then acts to set the heat level.

The controller must also accept updates to its target temperature with the `update_target_temperature` function. When there is no controller set, the Controller Manager will accept manual boiler level set commands. Manual commands of this nature are ignored when there is a controller set.

Code reference: src/controller/manager.rs

4.3.2.1. Boiler heat level

As mentioned above, the `sample` function returns a boiler heat level between `0` and `1`.

The boiler heat is controlled by the Solid State Relay that was described in the hardware section. The SSR is either on or off, and is turned on using the Pi's GPIO interface. When a GPIO pin is set *high*, it outputs 3.3V, when it is set *low* it outputs 0V.

The boiler's heating element is an electrical resistor and can generate a variable amount of heat by being fed a variable amount of current. I do not have the ability to vary the load to the boiler using the SSR, so I use Pulse-width modulation (PWM), a method that essentially toggles the power at a particular frequency, allowing me to emulate multiple power levels.

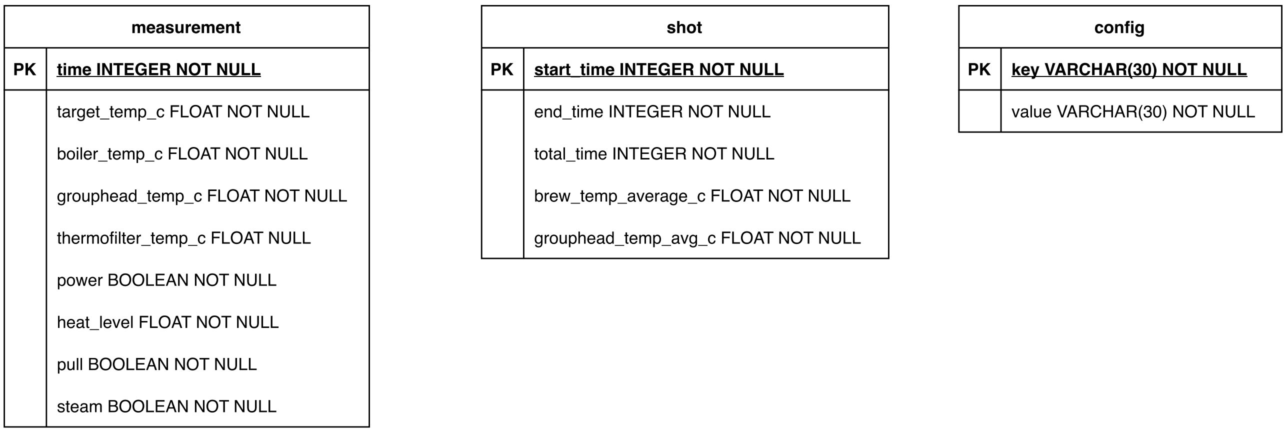
If I allow infinite precision (e.g. 0.01) it's possible for the duty cycle to be in a range that falls below the response time of the SSR (9ms), so I round the response of the `sample` function into a range of values between 0 and 1, with a step of 0.1 — thus supporting 10 levels of heat.

The period, or clock, of the PWM is 100ms, with the lowest heat level of 0.1 corresponding to 10ms. That is, with a heat level of 0.1, the heat is on for 10 out of 100ms; for a level of 0.5 it's on for 50 out of 100ms.

4.3.3. Database

The application uses an SQLite database to store three types of information:

- Configuration — Control Method, Target Temperature, UI settings, etc.
- Measurements — the boiler and grouphead temperatures at specific times.
- Shots — The time a shot was pulled, and when it ended. This is manually recorded via the UI, but is important for certain models.



measurement	
PK	time INTEGER NOT NULL
	target_temp_c FLOAT NOT NULL
	boiler_temp_c FLOAT NOT NULL
	grouphead_temp_c FLOAT NOT NULL
	thermofilter_temp_c FLOAT NULL
	power BOOLEAN NOT NULL
	heat_level FLOAT NOT NULL
	pull BOOLEAN NOT NULL
	steam BOOLEAN NOT NULL

shot	
PK	start_time INTEGER NOT NULL
	end_time INTEGER NOT NULL
	total_time INTEGER NOT NULL
	brew_temp_average_c FLOAT NOT NULL
	grouphead_temp_avg_c FLOAT NOT NULL

config	
PK	key VARCHAR(30) NOT NULL
	value VARCHAR(30) NOT NULL

Figure 14: Database table layout

As shown in Figure 14, the table layout is quite simple and the tables are all disjoint. The `shot` table's `start_time` and `end_time` can be used to query measurements by the `time` column using a range query, however, so they are related.

SQLite was chosen because the amount of data the application will generate does not warrant a dedicated DBMS, or the additional complexity with setting one up. The SQLite database is created and managed in Rust using `sqlx`, a tool that compiles SQL files into Rust types and also manages database schema migrations.

The schema files are stored in the `migrations` folder in the project root. Each migration contains a `.up.sql` and a `.down.sql` file, one to make a change (e.g. adding a table or altering a table to add a column), and the other to roll back the schema change.

SQL queries can easily be executed against the sqlite file over SSH, and there are examples of this in multiple model projects, which may contain a `query.sql` and an accompanying `query.sh` file, which executes a query against the Pi.

Code reference: src/core/db.rs

4.3.4. MQTT API

MQTT is a pub-sub protocol, and so does not have the concept of request-response messaging. MQTT has topics that can be subscribed and published to.

API methods fall into three categories:

1. *Get* — These methods provide a topic for getting their current value, the topic has the `retain` flag, so a subscriber will receive the latest value when they subscribe, they don't need to wait for the value to be updated. *Example: `gesha/temperature/boiler`*
2. *Get + Set* — This is the same as Get, but also provides a topic for setting the value. Messages published to the Set topic should not have the retain flag. *Example: `gesha/mode` and `gesha/mode/set`*
3. *Command* — A command method has a command topic that can be published to with no retain flag, and will contain a payload for the command. If the command has a response, there will be a response topic that must be keyed in the command payload, I use an `id` property. *Example: `gesha/temperature/history`*

Below is a complete API listing. I have omitted the `gesha/` prefix from the topic names to save space. All temperature values are in ℃. The payloads are shown here using TypeScript interface definitions, but should be serialised as JSON.

The full the API code can be found in `src/core/mqtt.rs`, and I have implemented a client in TypeScript `ui/src/geshaClient.ts` (for the UI) and Python `models/api.py` (for modelling and experiment automation).

Topic	Payload	Retain
temperature/boiler	{ timestamp: number; value: number }	true
temperature/grouphead	{ timestamp: number; value: number }	true
temperature/thermofilter	{ timestamp: number; value: number }	true
temperature/target	number	true
temperature/target/set	number	false
mode	"offline" | "idle" | "active" | "brew" | "steam"	true
mode/set	"idle" | "active" | "brew" | "steam"	false
control_method	"none" | "threshold" | "pid" | "predictive"	true
control_method/set	"none" | "threshold" | "pid" | "predictive"	false
boiler_level	{ timestamp: number; value: 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 }	true
boiler_level/set	0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0	false
config/{key}	string	true
config/set	{ key: string; value: string }	false
temperature/history/command	{ id: string; from: number; to: number; // The maximum number of measurements to be returned limit?: number; // The period in milliseconds for which to get a single measurement bucketSize?: number; }	false
temperature/history/command/{id}	Array<{ time: number; targetTempC: number; boilerTempC: number; groupheadTempC: number; thermofilterTempC?: number; power: boolean; heatLevel: 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0; pull: boolean; steam: boolean; }>	false
shot/history/command	{ id: string; from: number; to: number; limit?: number; }	false
shot/history/{id}	Array<{ startTime: number endTime: number totalTime: number brewTempAverageC: number groupheadTempAvgC: number }>	false

Table 3: MQTT API definitions

4.4. User Interface

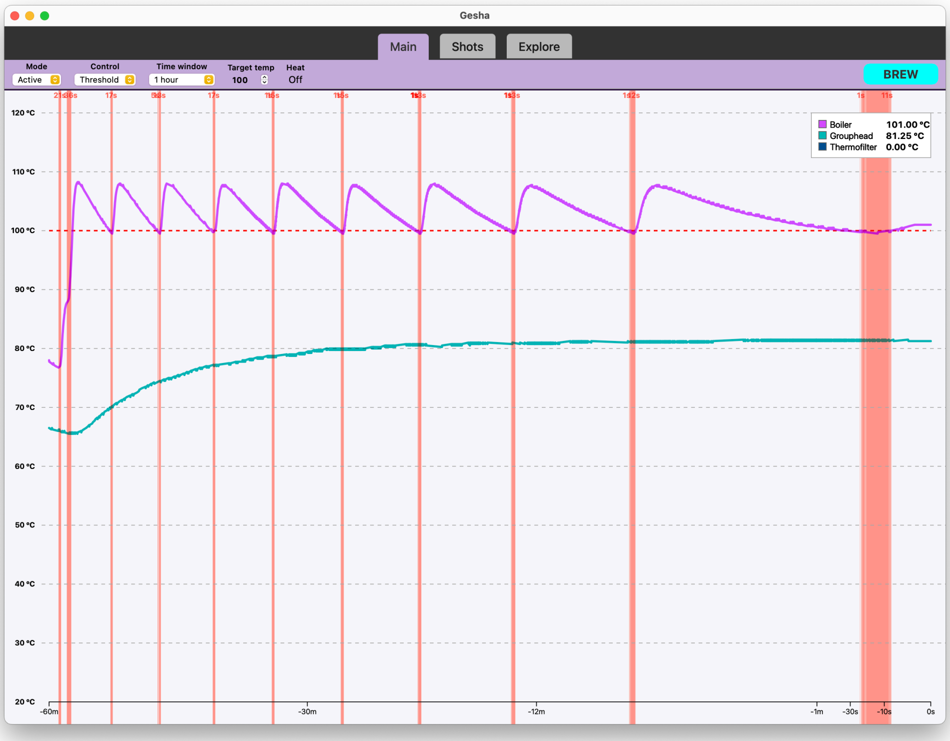
The user interface (UI) is used by the barista when brewing coffee, but also for looking up historic shot data and exploring measurement charts.

It is built with Web Platform technologies — HTML, CSS, and TypeScript (JavaScript but with types). The toolkit includes:

- SolidJS — a declarative framework for component views, similar to React
- mqtt.js — an MQTT client that uses WebSockets to connect the web app to the Mosquitto broker.
- D3 — a popular tool for creating SVG charts and graphs

4.4.1. Main tab

The Main tab (Figure 15) shows key information related to pulling a shot, including the current target temperature, boiler temperature, grouphead temperature, and the thermofilter temperature when it is available.



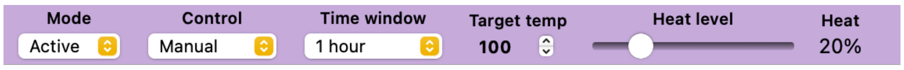
Mode: Active
Control: Threshold
Time window: 1 hour
Target temp: 100
Heat: Off

🟪 Boiler	101.00 °C
🟦 Grouphead	81.25 °C
🟦 Thermofilter	0.00 °C

Figure 15: Gesha — main tab showing a threshold controller at 100 ℃ over an hour period

The main chart is real-time and uses a square-root scale X axis, which results in the most recent measurements being the most visually prominent. For example, the screenshot shown in Figure 15 tells the barista that the machine has been maintaining its 100 ℃ target temperature using the stock threshold method since it was switched on about an hour ago. The red vertical lines indicate periods where the boiler was on and the water actively heating. When the temperature controller registers 100℃, the boiler is turned off. The thermal inertia of the water causes the temperature to rise a little more to approximately 108℃, before the heat starts to dissipate and the temperature drops. When the machine registers a drop below the target of 100℃, it turns the boiler back on and the pattern repeats. This is known as the "sawtooth graph". Notice that the grouphead holds its temperature better than the water in the boiler.

The control bar is located at the top of the main tab's interface, and allows changing the application's Mode, Control method, Time window (the X axis of the chart), and Target temperature. When the control method is set to manual, the option becomes available to set the heat level directly.



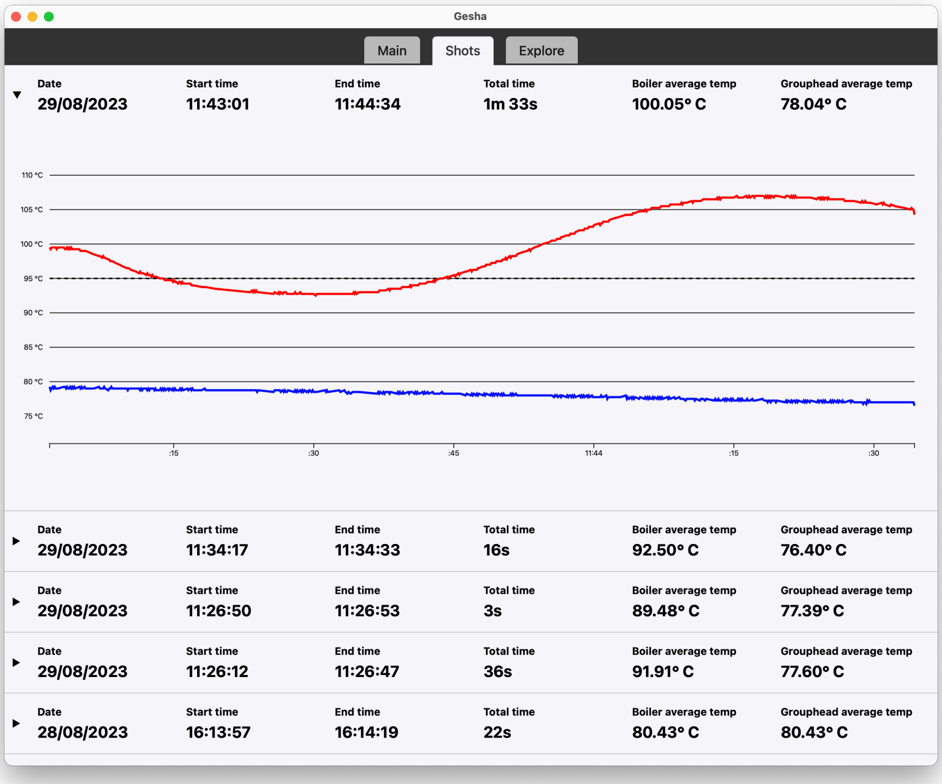
Mode	Control	Time window	Target temp	Heat level	Heat
Active	Manual	1 hour	100		20%

Figure 16: Manual control mode with heat level slider

Finally, the cyan *Brew* button on the right toggles the UI into Brew mode and starts a shot timer. The application records brew sessions and stores them in the brew table.

4.4.2. Shots tab

The shots tab (Figure 17) displays a listing of the `shot` table, with the option to expand a specific listing and show a chart of the measurements for that shot. From this pane, the barista can see that their last several shots have had a wide range of brew-times (from 3 to 90 seconds) but have been consistently brewed in the upper 70℃. The boiler temperature (red line) has been following the same sawtooth-pattern described on the homepage, while the grouphead's temperature has dropped slightly in the last hour. The chart is rudimentary, but useful. Note that the shots listed here were experiments to test the interface, and are not representative of real brewing data.



	Date	Start time	End time	Total time	Boiler average temp	Grouphead average temp
▼	29/08/2023	11:43:01	11:44:34	1m 33s	100.05° C	78.04° C
▶	29/08/2023	11:34:17	11:34:33	16s	92.50° C	76.40° C
▶	29/08/2023	11:26:50	11:26:53	3s	89.48° C	77.39° C
▶	29/08/2023	11:26:12	11:26:47	36s	91.91° C	77.60° C
▶	28/08/2023	16:13:57	16:14:19	22s	80.43° C	80.43° C

Figure 17: Gesha — shots tab showing an expanded entry with a chart

4.4.3. Explore tab

Lastly, the UI has a simple screen for showing a graph of measurements within a defined interval (Figure 18). This is useful for quickly visualising measurement data for a given time period. The characteristic sawtooth pattern is visible again, as well as a slight spike in temperature at approximately 11:15am when a shot was brewed and hot water passes through the grouphead. After that, the machine is switched off and the temperature drops back to the ambient room temperature.

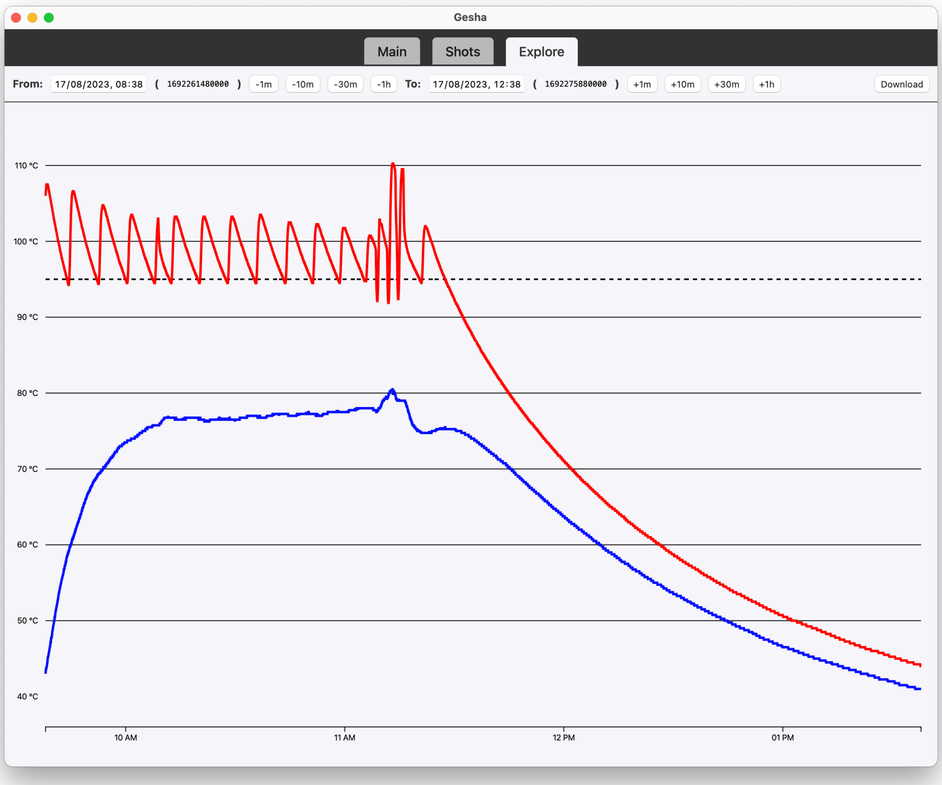

From: 17/08/2023, 08:38 (1692261480000) To: 17/08/2023, 12:38 (1692275880000)

Figure 18: Gesha — explore tab

5. Analysis

I have accumulated 5.5 million observations of the machine. When the machine is idle these measurements are taken every second, and when the machine is actively heating they are taken every 100ms, but only when the boiler or grouphead temperature changes. I use these observations to learn about the behaviour of the machine.

5.1. Pre-heating

As discussed in the introduction, the degree to which a machine has been pre-heated is a factor in many important aspects of the machine. In this section I will discuss the quantitative measure of determining the machine's pre-heating level, i.e. whether the machine has been fully pre-heated. This is an important factor to quantify, since it impacts the extraction temperature and therefore the taste of the espresso.

The machine could be said to be preheated when $\frac{T_{\text{Grouphead}}}{\max(T_{\text{Grouphead}})} = 1$, where $T_{\text{Grouphead}}$ is the grouphead temperature, but Figure 19 shows that the $T_{\text{Grouphead}}$ changes in response to the target temperature, and so will vary for all target temperatures. In the figure, you can see that the target temperature was changed from 105℃ to 95℃ at approximately 14:55. The machine recalibrates, and continues with its sawtooth pattern at the lower temperature.

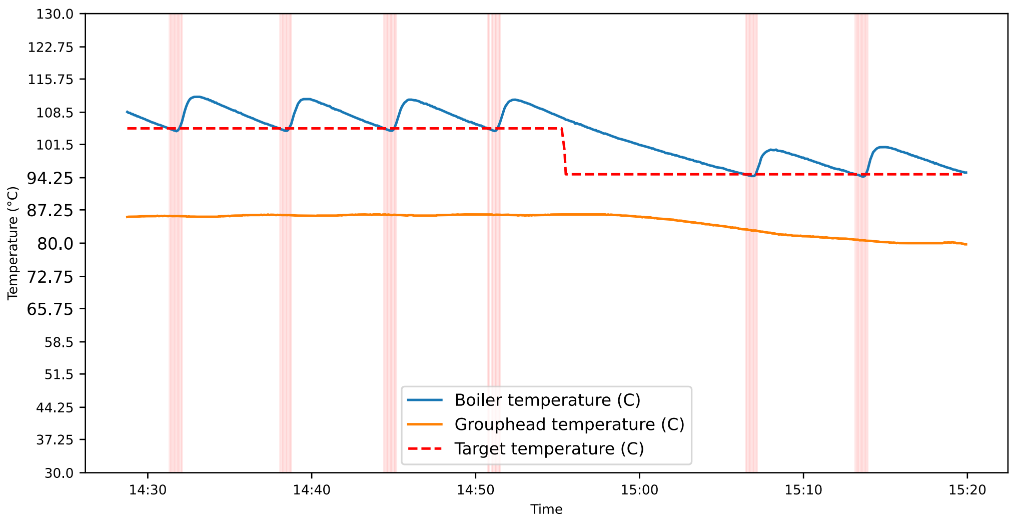


Figure 19: Grouphead (blue) and boiler (red) temperatures response to a target temperature change (dashed black)

Another reason that using the maximum value would produce inaccurate results is that the grouphead temperature can be very high when the machine is in steam mode, since T_{Boiler} is often > 140 ℃. Current database records show that the maximum value for $T_{\text{Grouphead}}$ is 108.5 ℃.

The heuristic I have chosen is to find a piecewise function that for each target temperature will return the equilibrium grouphead temperature. Notice in Figure 19 that $T_{\text{Grouphead}}$ stabilises over time. By taking the *modal* value of $T_{\text{Grouphead}}$ for each T_{Target} we can derive our piecewise function (Figure 20).

$$f(t) := \begin{cases} 74.0 \text{ if } t \leq 90.0 \text{ else} \\ 76.0 \text{ if } t \leq 93.0 \text{ else} \\ 78.0 \text{ if } t \leq 95.0 \text{ else} \\ 80.0 \text{ if } t \leq 99.0 \text{ else} \\ 82.0 \text{ if } t \leq 101.0 \text{ else} \\ 84.0 \text{ if } t \leq 103.0 \text{ else} \\ 86.0 \text{ if } t \leq 107.0 \text{ else} \\ 88.0 \text{ if } t \leq 109.0 \text{ else} \\ 90.0 \end{cases}$$

Figure 20: $f(t)$, where f is the heat level function and t is the target temperature

The piecewise function can be used to determine that the machine has fully pre-heated with the following:

$$\min\left(\frac{T_{\text{Grouphead}}}{f(T_{\text{Target}})}, 1\right) = 1$$

The function $f(t)$ will become more accurate over time, since its coefficients are derived by querying the measurement database (Listing 1). As such, re-running the query and updating the coefficients as more target temperatures are selected will further refine the coefficients.

```sql
WITH RoundedGroupheadTemperatures AS (
    SELECT
        target_temp_c,
        ROUND(grouphead_temp_c / 2) * 2 as rounded_grouphead_temp_c
    FROM measurement
    AND power IS TRUE AND STEAM IS FALSE AND pull IS FALSE
),

TemperatureCounts AS (
    SELECT
        target_temp_c,
        rounded_grouphead_temp_c,
        COUNT(*) as count
    FROM RoundedGroupheadTemperatures
    GROUP BY target_temp_c, rounded_grouphead_temp_c
),

ModalTemperatures AS (
    SELECT
        target_temp_c,
        MAX(count) as max_count
    FROM TemperatureCounts
    GROUP BY target_temp_c
)

SELECT
    m.target_temp_c,
    t.rounded_grouphead_temp_c as modal_grouphead_temp_c
FROM ModalTemperatures m
JOIN TemperatureCounts t ON m.target_temp_c = t.target_temp_c AND m.max_count = t.count
ORDER BY m.target_temp_c;
```

Listing 1: SQL query to find the modal $T_{\text{Grouphead}}$ for T_{Target}

It is also possible to rely on simply measuring how long the machine has been heating, however this method is not effective in a scenario where the machine is pre-heated fully for the first session and partially cooled for a second session. The pre-heating time will be shorter when the machine has not fully cooled down.

5.2. Boiler Power vs Temperature Increase

The threshold control method (descibed in the Introduction) yields poor results with regards to the accuracy of the temperature for two main reasons: the temperature reading from the boiler thermocouple does not reflect the true temperature of the boiler, it is attached to the outside case of the boiler, and not directly to the element itself. As such there is a lag between the application of heat in the boiler and the measurement of the results of this heat.

In this model I gather data that captures the relationship between the application of heat and the temperature change that we read. I will show that the pre-heat level of the machine is a dependent variable.

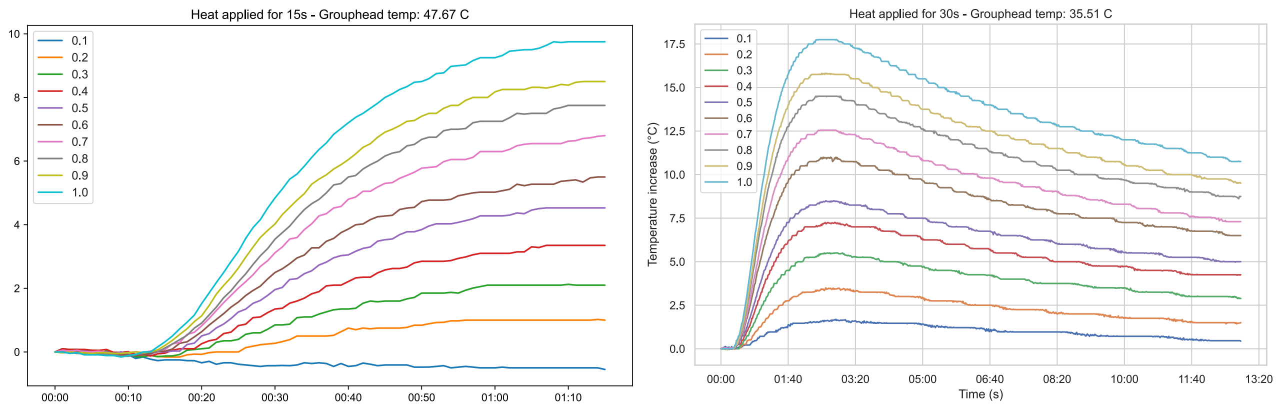


Figure 21: Temperature response to heat levels 0 — 1

5.3. Temperature Loss

The machine will cool when the boiler is not engaged. A component of the predictive model is the modelling of temperature loss, so that an appropriate heat time and duration can be executed to counteract the loss of heat. I define a query (Listing 2) to retrieve measurements from the machine as it cools down. I will use the results as a basis for a multiple linear regression model that can predict the future temperature in the absence of heating.

```
SELECT time, boiler_temp_c, grouphead_temp_c
FROM (
    SELECT time,
        LAG(boiler_temp_c) OVER () boiler_temp_c_prev,
        LAG(time) OVER () time_prev,
        boiler_temp_c,
        grouphead_temp_c,
        LAG(grouphead_temp_c) OVER () grouphead_temp_c_prev,
        heat_level,
        pull,
        power
    from measurement
) AS inner_query
WHERE boiler_temp_c_prev ⩾ boiler_temp_c
    AND grouphead_temp_c_prev ⩾ grouphead_temp_c
    AND heat_level = 0.0
    AND pull = FALSE
    AND power = FALSE
ORDER BY time ASC;
```

Listing 2: SQL query to retrieve measurements where the machine is cooling down

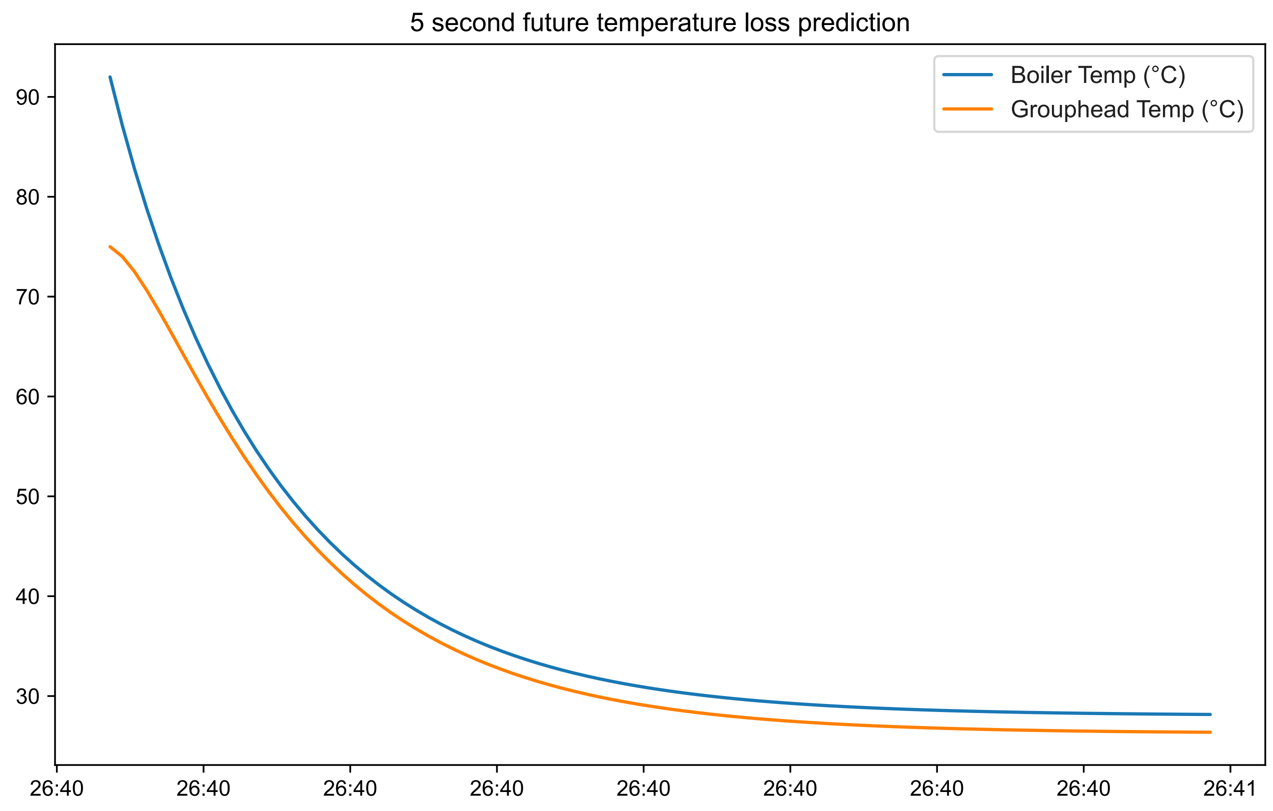


Figure 22: Temperature response to heat levels 0 — 1

The model takes an input of the current boiler and grouphead temperature and produces a predicted temperature loss for each after 5 minutes.

$$f(T_{\text{Boiler}}, T_{\text{Grouphead}}) = \Delta(T_{\text{Boiler}}, T_{\text{Grouphead}})$$

I arrived at 5 minutes as a time period by comparing the performance of 9 models, using the periods: $10s, 30s, 1m, 5m, 10m, 20m, 30m, 1h$. The models were trained using cross-validation with $\frac{2}{3}$ training data and $\frac{1}{3}$ testing.

The larger the time interval, the better the R^2 and MSE tended to be. However, the shorter the time interval is, the more useful the predictive model is. I found the 5 minute model was a good compromise between utility and accuracy.

Figure 22 shows the results of the 5 minute model predicting temperature loss with initial parameters of $T_{\text{Boiler}} = 92℃, T_{\text{Grouphead}} = 75℃$.

5.4. Heat sessions

There are a large number of measurements (more than 6 million), most of which are not useful for making predictions of the extraction temperature because they record the machine in an idle or cooling state. However, the identification and categorisation of these measurements is important for understanding the behaviour of the machine. A heat session is a discrete period of time where the boiler is active and heating. The period of time in between the application of heat and the temperature rise caused by this heat it reaching its peak defines the bounds of a heat session.

I implemented a multi-step procedure for identifying a range of measurements that belong to a specific heat session. Note that if two heat sessions are close together they will be merged, which muddies the analysis.

Contiguous measurement groups

There are occasional breaks in the heat session measurements (a discontinuity). This may be where the software was restarting, a bug in the software, or a loss of power. The first step is to identify contiguous groups of measurements in order to avoid incorrect analysis caused by a discontinuity. Measurements are only recorded when there has been a change in the boiler or grouphead temperatures. The sensors are read from every 100ms, this means that it is possible to rescale the recorded measurements to a precision of 100ms by back-filling the value from the previous measurement, since I know that the temperature has not changed in the intervening period. This presents a problem though, as there may be discontinuities where measurements were not recorded for reasons other than the temperature being stable.

$$\left|T_{\text{Boiler}_i} - T_{\text{Boiler}_{i+1}}\right| > 2.0℃$$

Figure 23: The last measurement in a contiguous group is determined when the temperature difference to the next measurement is more than 2 ℃

I analysed the measurements and determined that a temperature rise of 2 degrees is possible within a 100ms period, but any higher than this is a discontinuity. It is not as reliable to apply such a heuristic directly to the timestamp because when the temperature is stable during idle, it is possible that no measurements are recorded for large periods (hours), making it impossible to determine discontinuity using the timestamp.

Code reference: predictive/dataset.py#group_by_contiguous_measurements

Heat session bounds

For every contiguous group of measurements, I must find the subset of measurements where heat was being applied and the temperature consequently increased. To determine the start of a heat session I find the indices for measurements where the heat level (a value between 0 and 1, with intervals of 0.1, indicating the level of heat the boiler is outputting) goes from 0 to >0, followed by measurements remain >0 within a threshold. The threshold is defined because there are periods of time where the heat level will osculate between 0 and 1 and this does not indicate the heat session has ended. There is a single iteration through the measurements to determine the bounds, this prevents heat sessions from overlapping.

Code reference: predictive/dataset.py#get_heat_session_bounds

Start lag and **Stop lag**

When the boiler heating element is engaged, the heat is instantly applied to the water in the boiler. However the resulting temperature rise does not propagate to the boiler temperature sensor for some period of time - this lag time is called the start lag. To measure the start lag we measure the time between the start of the heat session and the first measured temperature increase.

Similarly, when the heating element is disengaged the temperature does not immediately stop rising due to accumulated thermal inertia - I call this the stop lag. I measure the stop lag as the amount of time between the heat level going to 0 and the boiler temperature reaching a plateau.

Code reference: predictive/dataset.py#get_heat_session_summaries

Heat session anatomy

Heat sessions are not always easy to separate from the measurement data, so I define some characteristics that identify bad data. Heat sessions with an instant start lag indicate that there was already thermal inertia that caused the temperature to increase, making this heat session inaccurate. Sessions with an immediate stop lag, or an exceedingly long stop lag are also excluded. Finally, sessions with a starting boiler temperature of $>100℃$ are excluded. Figure 24 shows the ratios for a mean average heat session. The colored bar represents the stats of the boiler, while the dotted markers indicate where the output data registers a change.

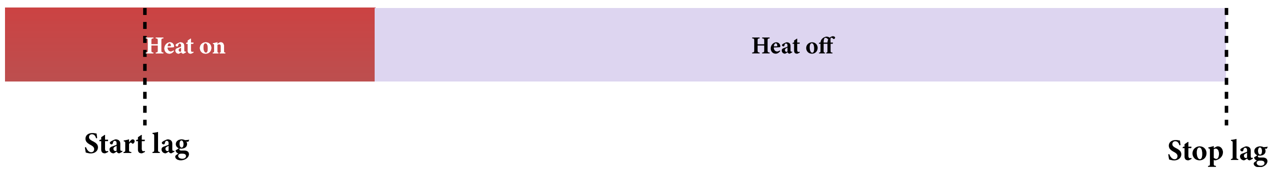


Figure 24: Heat session anatomy

Table 4 shows summary statistics for the heat sessions. Further work could be done to remove outliers, including the performance of automated experiments to generate measurements that are easier to separate. It is important to have a reasonably large sample of observations, however, as such the choice of parameters for culling outliers is conservative.

	Session duration (seconds)	**Total heat** (seconds)	**Start lag** (seconds)	**Stop lag** (seconds)	**Starting boiler temperature** (℃)	**Temperature** Δ (℃)
count	533	533	533	533	533	533
mean	93.37	21.20	7.95	48.74	93.33	10.78
std	14.45	16.10	5.58	9.74	9.73	8.63
min	37.10	1.78	2.00	25.40	24.25	0.75
25% (Q1)	87.40	15.44	3.00	43.30	90.00	8.25
50% (Q2)	91.30	16.89	5.30	46.20	95.00	9.00
75% (Q3)	96.40	20.24	13.50	50.50	100.00	10.75
max	214.30	130.39	19.90	89.40	100.00	81.25

Table 4: Summary statistics for the heat sessions

5.5. Extraction Temperature

Extraction Temperature ($T_{\text{Extraction}}$) is the temperature of the water as it comes out of the espresso machine and into the portafilter. This is the temperature that I am interested in, not the boiler, because it is the temperature of the water as it meets the coffee and subsequently the temperature that has an effect on the coffee's taste.

Because observations are time consuming to gather there are relatively few observations available. My process for gathering observations is as follows:

1. Lock the thermofilter into the grouphead,
2. Actively record thermofilter measurements on the secondary Pi Zero, recording a timestamp and the current grouphead temperature
3. Tap the Gesha UI brew button and flip the brew switch simultaneously,
4. Observe the thermofilter measurements in the UI — they are broadcast from the secondary Pi Zero over MQTT and integrated into the UI
5. Flip the brew switch back when the thermofilter temperature dovetails with the boiler temperature

I follow this procedure repeatedly, varying the target temperatures and the pre-heat level. The thermofilter is emptied and the grouphead is flushed between sessions. I combine all observations and filter them by the shot `start_time` and `end_time` (Figure 25). I have 41 experiments, and a combined total of $10,043$ observations.

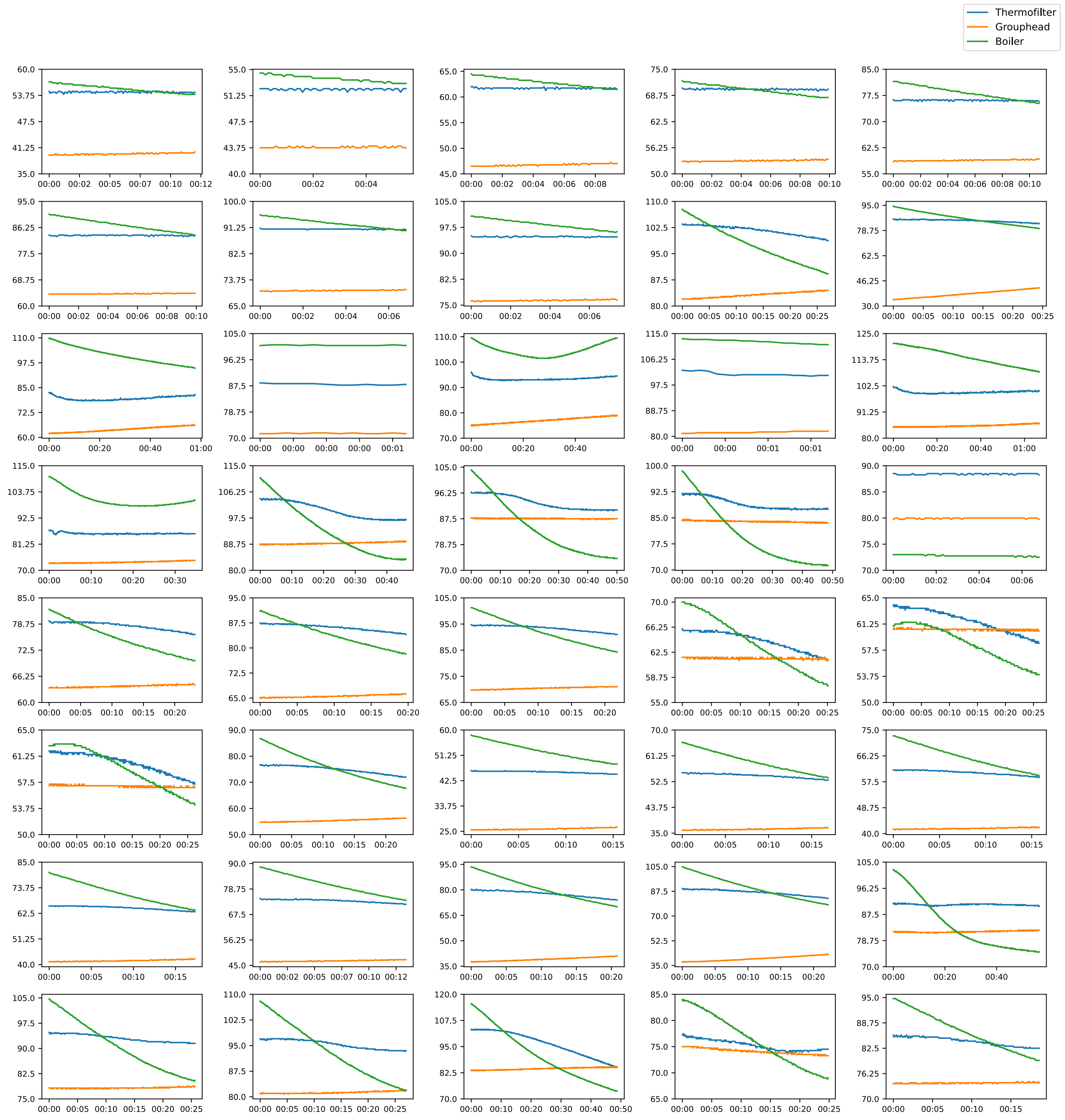


Figure 25: All extraction temperature measurements

The Pearson correlation coefficient (r) is a measure of the linear relationship between two variables. The coefficient has a range of -1 to 1, where 1 or -1 are a perfect (positive and negative, respectively) linear relationship between two variables and 0 represents variables with no correlative relationship. Figure 26 shows a heatmap of the coefficients for $T_{\text{Grouphead}}$, T_{Boiler}, and $T_{\text{Extraction}}$. The r with respect to $T_{\text{Grouphead}}$ and $T_{\text{Extraction}}$ is 0.8, and r with respect to T_{Boiler} and $T_{\text{Extraction}}$ is also 0.8.

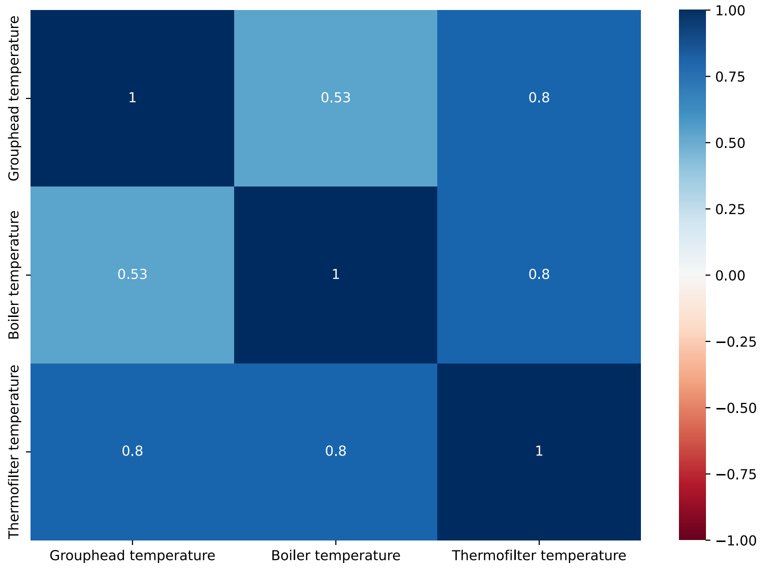


Figure 26: Pearson correlation heatmap for $T_{\text{Grouphead}}$, T_{Boiler}, and $T_{\text{Extraction}}$

An r value of 0.8 suggests a moderate to high linear correlation with the extraction temperature. Each model was trained with 80% of the data, with 20% reserved for testing. More data in more varied conditions will improve the model further, and this model in particular will benefit from more accuracy.

First I fit a multiple linear regression model using $T_{\text{Grouphead}}$ and T_{Boiler} to predict $T_{\text{Extraction}}$. This produced a model with $R^2 = 0.843$ and a $\text{MSE} = 31.5$, $\sqrt{\text{MSE}} = 5.5$℃.

Next I trained a Support Vector Machine (SVM) with the Radial Basis Function to test if a non-linear regression performed better. I used a Grid Search Cross Validation and found the coefficients $\gamma = 1.0$ and $C = 1000$ to perform the best. Training a model with these parameters resulted in a better fitting model with $R^2 = 0.935$ and $\text{MSE} = 13$, $\sqrt{\text{MSE}} = 3.6$℃.

Finally, I trained a neural network using Tensorflow, starting with a single layer network with 16 units and experimenting with the number of units and hidden layers until a well-performing model was produced. I used a tanh activation function and a linear output. The training data was not rescaled because the inputs and outputs are all in the same unit (degrees celsius). I trained the model using an MSE loss function.

I produced a simple neural network model with $R^2 = 0.96$ and $\text{MSE} = 7.6$, $\sqrt{\text{MSE}} = 2.75$℃ (Figure 27). This is the best performing model. I integrate this model into the Gesha application, publishing a predicted value on the `temperature/thermofilter_predicted` topic. This is rendered in the UI in real time along with the boiler and grouphead temperatures, providing a useful prediction to the barista.

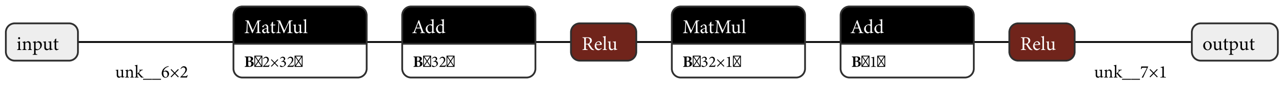


Figure 27: A graph of the extraction temperature neural network

6. Inference on the Pi Zero

Machine Learning models are typically run in production using the same software stack that they were trained with (e.g. Python + Tensorflow, PyTorch, XGBoost, etc). Containers are often used to make this process simpler and less error prone. My application has a unique set of constraints: the model must produce an inference result within 100ms whilst running on a low-power device (the Pi Zero has a 1GHz single core CPU and 512MB RAM), so that the information can guide a barista in real-time. Whilst there are container runtimes under Linux with very low overhead, with runc being a good choice [37], there are additional complications with software availability for 32-bit ARM. Furthermore, I benchmarked the time required run a "hello world" script on the Pi Zero with no other software running — it took 272ms.

To account for these constraints, I looked into running the model within the Rust application itself. This would provide a large speed improvement since Rust is close to C in terms of performance, and also because any overhead incurred by starting a sub-process will be avoided. The Open Neural Network Exchange (ONNX) [38] is a format that models the network as a directed acyclic graph (DAG). The nodes in the graph represent specific mathematical operations from simple addition or matrix multiplication to long short-term memory (LSTM) recurrent neural networks. Edges describe the inputs and outputs between these nodes, and "initialisers" can be used to store data, like trained weights and biases, within nodes.

It is possible to convert a Tensorflow model to ONNX, which is serialised using protocol buffers (protobuf) [39], a binary format invented at Google. In order to run inference with the model, an ONNX runtime is required. The runtime will parse the protobuf file into the ONNX DAG. The runtime will need to implement the operators that are defined in the ONNX specification and allow the user to pass in data to the input node, and return the result produced in the output node of the network.

Tract is an ONNX runtime, or "inference engine" that is developed by Sonos [40], originally to support wake word recognition on its low-performance speakers. Tract is implemented in Rust, which makes it ideal for this use case. An alternative runtime that implements a larger set of operators is ONNX runtime. I chose not to use this because it is a C++ library and does not have an official Rust module, which makes integration more involved.

The downside to this approach is that models converted from Tensorflow to ONNX do not always work, either due to a lack of operator support in the runtime (meaning the model won't run at all) or a difference in the weight types (`float32` vs `float64`, for example) resulting in errors in inference (the inference result is an order of magnitude different from the Tensorflow inference result). As such I have kept my models as simple as possible, relying on optimisation of training data instead of more complex models.

7. Predictive Model

As I have shown in the heat session analysis, the boiler temperature will continue rise for some time after boiler heat is no longer applied (stop lag). The amount of temperature increase is related to the amount of time the boiler was heating for, and the degree to which the machine was pre-heated.

Each measurement has the following features:

- `time` - the time in UNIX epoch milliseconds
- `target_temperature` - the target temperature in degrees celsius
- `boiler_temperature` - the temperature of an espresso machine's boiler, read from a thermocouple probe mounted to the exterior of the boiler
- `grouphead_temperature` the temperature of the grouphead, the metal block to which the portafilter will be locked, which is used as a proxy variable for the machine's level of pre-heating
- `heat_level` - A number between `0` and `1`, with a step of `0.1`, which represents the amount of heat being applied to the boiler

Measurements are taken at different intervals depending on the machine's state. During an active heating session measurements are taken every 100ms, but in the idle mode records are recorded only every 1s. I used the following equation to calculate the thermal inertia carried by the water, where I is thermal inertia, Q is the energy input, and ΔT is the rate of temperature change.

$$I = Q\Delta T$$

I calculate Q for each measurement by summing the heat level over a time period, which I define as being the stop lag time. In theory the stop lag time is a measure of how long the heat affects the temperature rise, and therefore in our estimate for Q we should include all heat levels in that time window. A new feature, `heat_level_sum` is added to each measurement, defined as a rolling sum of heat levels for the previous records up to a maximum time interval derived from the stop lag time.

```
measurements["heat_level_sum"] = (measurements["heat_level"]
    .rolling(window=m, min_periods=1)
    .sum()
    .fillna(0)
)
```

I then derive a new feature `future_boiler_temperature` by subtracting the boiler temperature n measurements into the future from the current boiler temperature, creating a feature that contains the future temperature difference.

```
measurements["boiler_temp_c_future"] = measurements["boiler_temp_c"].shift(-n)
```

I define a simple neural network using TensorFlow — a commonly used library for training neural networks — to predict `boiler_temp_c_future` from `grouphead_temp`, `boiler_temp`, and `heat_level_sum`. I define a model with 2 hidden layers with 64 and 8 units respectively. I use the tanh activation function for the hidden layers and a linear activation for the output. I train for 50 epochs and a batch size of 128. There are 46 million observations total, I reserve 33% for testing the model.

In this first model I use $m = 260$ and $n = 2160$, which were not derived from the stop lag time. Instead, these coefficients were found by searching permutations of m and n such that the Pearson correlation coefficient between `heat_level_sum` and `boiler_temp_c_future` was the highest.

The fit model has a mean squared error (MSE) of 49 and an $R^2 = 0.75$. The model is not very well fit, but may show interesting results. I convert the model to ONNX format (visualised in Figure 28) and validate the inference results are consistent between Tensorflow and the ONNX runtime.

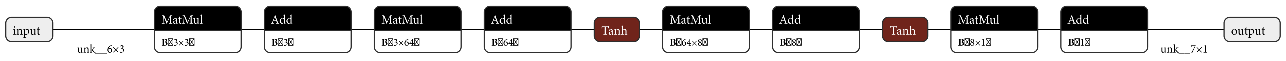


Figure 28: A graph representation of the neural network exported to ONNX

To integrate the ONNX model into the Rust application code, I implement a module to expose the prediction function `predict_boiler_temp_diff`. I update the controller manager to keep a rolling sum of heat levels and update the definition of the `sample` function to include the rolling sum.

I implement the `sample` function for the predictive control method:

```
fn sample(&mut self, boiler_temp_c: f32, grouphead_temp_c: f32, heat_level_sum: f32) → f32 {
    let predicted_boiler_temp = self.model.predict_boiler_temp_diff(
        grouphead_temp_c,
        boiler_temp_c,
        heat_level_sum,
    );

    ...

    let heat_level = if predicted_boiler_temp > self.target_temperature {
        0.0
    } else {
        1.0
    };

    heat_level
}
```

Figure 29 shows a real-world test of the model. The model is attempting to hold the boiler temperature at 100℃ over a 2.25 hour period between 11:30am and 2:15pm. The red vertical bars represent where the model turned on the boiler, whose temperature is represented by the blue line. The orange line represents the grouphead temperature, and the red dashed line the target temperature. As apparent, the prediction is not stable. Although it accurately identifies points where the non-heating water will decrease in temperature, it does not allow the temperature drop even when the water is higher than the target. In these cases, the model turns the boiler on anyway to combat the temperature drop, and just moves the boiler temperature further away from the target. It also has periods were it allows the temperature to drop well below the target without a rapid attempt to correct it. Notice too that the grouphead temperature has much more temperature variation here than it did in the threshold control method described earlier.

Red vertical lines indicate the boiler heat being turned on. The predictions are not stable, and often predict a temperature drop despite the temperature being well above the target. The model does keep the temperature relatively close to the target temperature (shown by the red horizontal dashed line), but would also drive the heat very high (see around the 14:00 interval).

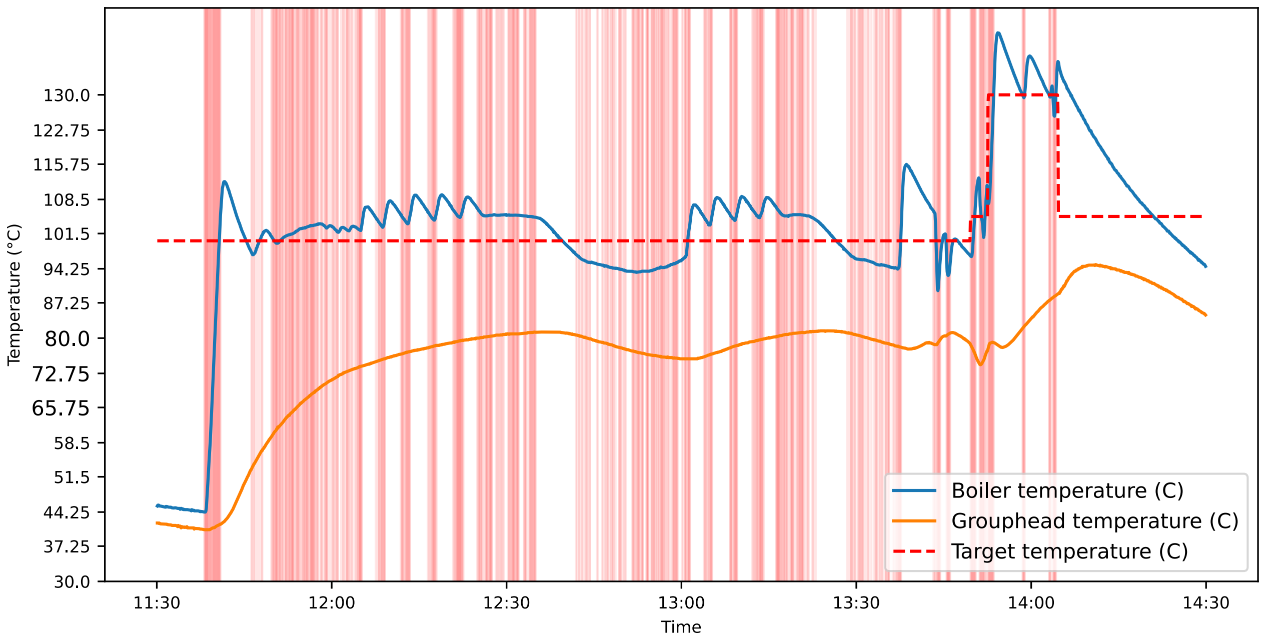


Figure 29: Predictive control test

I theorise that because the model was trained on the entire measurement set, and the temperature decreases for the majority of observations (as the machine is idle or cooling most of the time), the model has fit on these measurements instead of the more important heat session measurements.

My next steps for producing a more reliable model would be to fit a new model only on measurements that are within a heat session, with a time margin before and after so that the heat level sum has a gradual increase. I would do more analysis into heat sessions in order to remove sessions that are not a good representation of the relationship between heat output and temperature difference. Ensuring the quality of the training data whilst also retaining enough training data to fit a model that will generalise is possible, but may require more measurements. I have implemented a Python API that enables the automation of experiments, which I originally used in my analysis of the boiler heat level responses. (Code reference: `experiments/repeatable/power_level/__main__.py`). Further such experiments will help to produce more self-contained measurement samples that can be used for model training. Ideally all measurement data can be utilised, and work will be undertaken to filter out low quality measurements.

By augmenting the measurement data with the `heat_level_sum` and `future_boiler_temperature` features I have manually created new features that I had hoped would capture the desired relationship. This may have been too simple, and the relationship may be learnt directly instead of through manual intervention. If the model produced from the heat session measurements also performs poorly I will explore a time series long short term memory (LSTM) architecture that may be able to capture the relationship between the heat input and temperature rise. Such an architecture could fit on patterns in time series data, which our measurement data undoubtedly is. This could allow a model to be trained on a series of measurements for an interval, and find patterns within this series. A larger model could also include more features such as `brew` and `steam`, as well as `boiler_temperature`, `grouphead_temperature` and `heat_level`.

Such an architecture would present new challenges for integrating the model into the main application though, as the reason a simple model was initially chosen was to make inference easy and fast. An LSTM model necessitates a 3-dimensional structure that I have experimented with for a new extraction temperature model, but have had difficulty integrating into the main application because of the more complicated conversion to ONNX format. As well as integrating the model and performing inference the software will have to store a large number of measurements in memory to be used for model input. This additional overhead may cause problems for sub-100ms inference on the Pi.

8. Conclusion

Although the performance of the predictive model is not good enough to use practically, it demonstrates that the temperature can be held relatively stable with this approach. I am confident that with further iteration a predictive control model that is stable and substantially better than the threshold control method will emerge.

Upon completion of this project, the espresso machine now possesses advanced functionalities akin to a smart device. It can be operated remotely, allowing for power control and temperature adjustments. Real-time monitoring is available for both boiler and grouphead temperatures, alongside an accurate estimate of the extraction temperature. Additionally, there's a feature to review historical data, detailing previous shots and their respective average boiler temperatures during brewing sessions.

I have implemented the control manager such that new control methods are simple to add. Some methods that I would like to explore include a fuzzy logic system that can be defined using the known parameters of the system, such as start and stop lag time, and heat level vs temperature rise estimates. A PID neural network [41] trained using a simulated Silvia based on my dataset, which has grown to include 6,234,653 measurement observations, is a possibility.

I would also like to explore further hardware modifications, including an RTD sensor that can be mounted in a lower latency zone of the boiler [42], as I would like to eliminate any additional latency caused by the type K thermocouple or the insulating silicone pad. The aforementioned Pi 2 is slowly becoming available which opens up new possibilities, especially as the software is implemented with asynchronous programming paradigms, it will be ready to take advantage of multiple cores.

Software projects are never finished, and I plan to continue the development of this project until I have a predictive model that can hold the temperature stable. I have found this project a worthy challenge. Cheers .

Bibliography

[1] Instituto Nazionale Espresso Italiano, *The Certified Italian Espresso and Cappuccino*. [Online]. Available: http://www.espressoitaliano.org/files/File/istituzionale_inei_hq_en.pdf

[2] J. Hoffmann, *The World Atlas of Coffee, Second Edition*, Mitchell Beazley, 2018.

[3] F. Brookshier, *The Burro*, University of Oklahoma Press, 1973.

[4] M. Gordon, “The Coffee Story: Monks, Myths and Goats.” [Online]. Available: https://www.chaldeannews.com/features-1/2021/7/29/the-coffee-story-monks-myths-and-goats

[5] K. Seidel, “Coffee - The Wine of Islam: From Serving the Guest: A Sufi Cookbook & Art Gallery.” [Online]. Available: https://www.superluminal.com/cookbook/essay_coffee.html#:~:text=The%20beverage%20became%

[6] V. H. - The Coffee Guru, “Turkish sand coffee: discover its history, preparation, and cultural significance.” [Online]. Available: https://thecoffeeguru.net/turkish-sand-coffee/#origins-of-turkish-sand-coffee

[7] G. Linguistica, "Espresso etymology." [Online]. Available: https://www.garzantilinguistica.it/ricerca/?q=espresso#:~:te

[8] M. A. Moriondo, "Nuovi apparecchi a vapore per la confezione economica ed istantanea del caffè in bevanda. sistema a. moriondo." [Online]. Available: https://commons.wikimedia.org/wiki/File:Espresso-machine-first-patent-angelo-moriondo.jpg

[9] B. Hustle, "Angelo Moriondo - The World's First Espresso Machine Revealed." [Online]. Available: https://www.youtube.com/watch?v=JELm1GePFP4

[10] G. Milano, "Our History." [Online]. Available: https://www.gaggia-na.com/pages/history#:~:text=Achille%20Gaggia%

[11] S. Eats, "Coffee Jargon: 'Pulling' a Shot." [Online]. Available: https://www.seriouseats.com/coffee-jargon-why-is-it-called-pulling-a-shot-espresso-barista-terms

[12] J. Stamp, "The Long History of the Espresso Machine." [Online]. Available: https://www.smithsonianmag.com/arts-culture/the-long-history-of-the-espresso-machine-126012814/#:~:text=no%20coffee

[13] Wikipedia, "E-61." [Online]. Available: https://en.wikipedia.org/wiki/E-61

[14] S. C. Association, "What is specialty coffee?." [Online]. Available: https://sca.coffee/research/what-is-specialty-coffee

[15] S. Loofbourow, "Understanding extraction," *Roy. Coffee*, 2017. [Online]. Available: https://royalcoffee.com/understanding-extraction/#respond:~:text=Only%2030%25%20of%20a%20coffee%20bean%20is%

[16] B. Hussle, "What difference will it make to an espresso." [Online]. Available: https://www.baristahustle.com/blog/what-difference-will-it-make-to-an-espresso-if-i-raise-the-temperature-of-my-machine-by-one-degree/#:~:text=For%20lighter%20roasted%2C%20special%20coffee%2C%20t

[17] C. A. Salamanca, N. Fiol, C. González, M. Saez, and I. Villaescusa, "Extraction of espresso coffee by using gradient of temperature. effect on physicochemical and sensorial characteristics of espresso," *Food Chemistry*, vol. 214, pp. 622–630, 2017, doi: https://doi.org/10.1016/j.foodchem.2016.07.120. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0308814616311530

[18] S. Andueza, L. Maeztu, et al., "Influence of water pressure on the final quality of arabica espresso coffee. application of multivariate analysis," *J. Agricultural Food Chemistry*, vol. 50, no. 25, pp. 7426–7431, 2002.

[19] S. Andueza, M. A. Vila, M. Paz de Pe~na, and C. Cid, "Influence of coffee/water ratio on the final quality of espresso coffee," *J. Sci. Food Agriculture*, vol. 87, no. 4, pp. 586–592, 2007.

[20] pidsilvia.com, "PID control for Rancilio Silvia." [Online]. Available: http://www.pidsilvia.com/

[21] auberins.com, "PID Temperature Control Retrofit KIT for Rancilio Silvia [KIT-RSRTDNb]." [Online]. Available: https://www.auberins.com/index.php?main_page=product_info&products_id=36

[22] "MeCoffee pid kit." [Online]. Available: https://mecoffee.nl/mecoffee

[23] jonk. [Online]. Available: https://electronics.stackexchange.com/questions/311507/please-explain-in-laymans-terms-how-a-pid-accounts-for-inertia-in-temperature-c#:~:text=There%20is%20nothing,delay

[24] [Online]. Available: https://mecoffee.nl/wp-content/uploads/2014/03/graph.svg

[25] S. Pennisi, "Pandemic, shortages, and electronic engineering," *IEEE Circuits Syst. Mag.*, vol. 22, no. 3, pp. 41–49, 2022.

[26] Shelly, *Shelly 1 Relay Manual*. [Online]. Available: https://shellystore.co.uk/wp-content/uploads/2021/05/Shelly-1-Specifications.pdf

[27] "Fotek ssr-40 da datasheet." [Online]. Available: https://cdn.sparkfun.com/datasheets/Components/General/SSR40DA.pdf

[28] "Max31855 datasheet." [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/max31855.pdf

[29] home-barista.com, “Building a diy thermofilter.” [Online]. Available: https://www.home-barista.com/espresso-machines/building-diy-thermofilter-t38248.html

[30] “Espresso Machine Thermofilter 1 - Temperature Device - Scace Device Only.” [Online]. Available: https://www.espressoparts.com/products/espresso-machine-thermofilter-1-temperature-device-scace-device-only

[31] OASIS, “MQTT Specification.” [Online]. Available: https://mqtt.org/mqtt-specification/

[32] R. A. Light, “Mosquitto: server and client implementation of the mqtt protocol,” *J. Open Source Softw.*, vol. 2, no. 13, p. 265, 2017.

[33] O. F. & Contributors, “Node-red.” [Online]. Available: https://nodered.org/

[34] “Home Assistant.” [Online]. Available: https://www.home-assistant.io/

[35] Wikipedia, “Webhook.” [Online]. Available: https://en.wikipedia.org/wiki/Webhook

[36] WHATWG, “Html standard - server sent events.” [Online]. Available: https://html.spec.whatwg.org/multipage/server-sent-events.html

[37] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt, “Performance evaluation of container runtimes,” in *Closer*, 2020, pp. 273–281.

[38] J. Bai, F. Lu, K. Zhang, and others, “Onnx: open neural network exchange,” GitHub, 2019. [Online]. Available: https://github.com/onnx/onnx

[39] Google, “Protocol buffers.” [Online]. Available: https://github.com/protocolbuffers/protobuf

[40] Sonos, “Optimising a neural network for inference.” [Online]. Available: https://tech-blog.sonos.com/posts/optimising-a-neural-network-for-inference/

[41] H. Shu, and Y. Pi, “Pid neural networks for time-delay systems,” *Comput. & Chem. Eng.*, vol. 24, no. 2-7, pp. 859–862, 2000.

[42] “Pt100 RTD temperature sensor - M4 mounting screw for Gaggia and some Rancilio Silvia machines --- shadesofcoffee.co.uk.” [Online]. Available: https://www.shadesofcoffee.co.uk/pt100-rtd-temperature-sensor---m4-mounting-screw-for-gaggia-and-some-rancilio-silvia-machines