Exploring the Challenges of Device Integration within the Smart Home Ecosystem: A Case Study on Home Assistant

Conor Gallagher, Queen's University Belfast

***Abstract*—The last decade has witnessed a boom in the popularity of IoT smart home devices, with household penetration reaching 73.5% in 2024 and expected to hit 90.7% by 2029 [22]. However, with the sudden and rapid proliferation of this technology, software practitioners have not had ample time to fully understand the complexities of the smart home ecosystem, particularly in contributor-driven platforms such as Home Assistant. This project aims to comprehensively profile the challenges of developing device integrations to identify common pitfalls faced by developers and propose best practices for creating robust integrations.**

I. Introduction

'Smart homes' are becoming increasingly popular, allowing homeowners to control and automate elements of their home with ease. The integration of voice assistants like Amazon's Alexa and Google Assistant, combined with the rapid growth of smart devices such as thermostats, lighting, and security cameras, creates a seamlessly interconnected living environment. These smart home ecosystems not only enhance convenience and efficiency through remote control but also contribute to environmental sustainability with applications like smart thermostats. However, like any software/hardware systems, they are prone to malfunctions, data breaches, and privacy leakages via security attacks, all particularly impactful given the close relationship between users' homes and these devices. For these reasons, building a reliable and robust smart home ecosystem is of utmost importance.

A number of smart home platforms are currently available to consumers, including Samsung SmartThings, Google Home, and Apple HomeKit [17, 13, 11]. However, these commercial options are largely closed-source, and support only approved devices within their proprietary ecosystems. The available automations are often very limited, offering simplicity in exchange for reduced automation capabilities and advanced functionality. Moreover, these platforms typically require constant internet access to operate through the vendor's cloud service, raising concerns about privacy, reliability, and responsiveness. In response, open-source, community-driven platforms like Home Assistant [14] have gained popularity, addressing these shortcomings. Notably, Home Assistant's repository was the most contributed-to open source project in 2024 [9].

To interact with a smart device through Home Assistant, a **device integration** is needed, which enables the monitoring, control, and automation of the device based on its state. Device integrations are often substantial pieces of software, requiring adherence to strict patterns and responsibilities to ensure seamless collaboration with the broader Home Assistant platform. By analyzing pull requests for device integrations in the Home Assistant repository, we aim to answer two key research questions:

1) What challenges do developers face when creating device integrations?
2) What are the characteristics of device integration pull requests?

II. Background

Home Assistant brands itself as the 'open source home automation that places local control and privacy first'. Since its initial release in 2013, it has rapidly become a cornerstone of many smart homes worldwide. A key factor behind its widespread adoption is the flexibility and extensibility of its architecture. This includes the central **Core** module, whose behavior can be extended with **integrations**, and supplemented with **add-ons**.

A. Core Module

The main component of Home Assistant is the Core module [15], which features several key elements:

- **Event Bus**: Facilitates the firing and listening of events.
- **State Machine**: Tracks the states of entities and monitors for updates. An **entity** can be thought of as a data point that provides information to Home Assistant; for example, a physical device may have multiple sensors for humidity, temperature, and luminance, which will each be modeled as different entities.
- **Service Registry**: Executes actions based on detected events, enabling integrations to register services on the event bus.

B. Integrations and Add-ons

Beyond its core functionality, Home Assistant (HA) Core is extended through third-party Python libraries, namely add-ons and integrations. Add-ons allow users to enhance Home Assistant's capabilities by installing additional applications, such as MariaDB for database support or an MQTT broker. These add-ons run alongside HA Core in separate Docker containers, leveraging **Home Assistant OS** (**HAOS**)- a lightweight, embedded operating system designed to run HA on a variety of

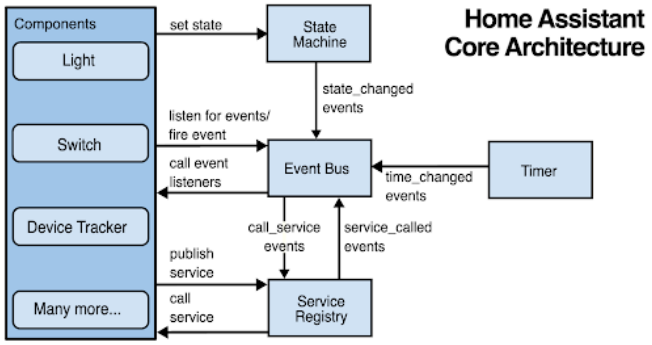

Figure 1: Home Assistant Core Architecture [15]

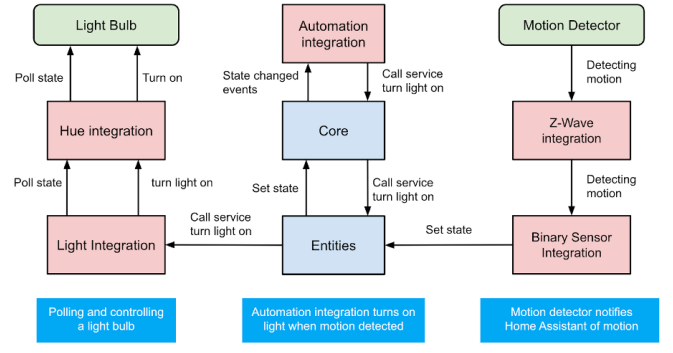

Figure 2: Interactions within Home Assistant [16]

devices, most popularly Raspberry Pis. HAOS is the recommended installation type for Home Assistant, although other options, such as running HA Core in an isolated container, are also available.

Integrations, on the other hand, enable functionality to be integrated directly into HA's core. Home Assistant defines four categories of integrations [16], with the three most significant being:

- **Entity Integrations**: These integrations model specific categories of IoT devices and define the data available to Home Assistant, thus serving as platforms upon which device-specific integrations can be built. For example, the *light* integration defines data attributes like `brightness` and `color_mode`, along with actions such as `turn_on`, which are exposed to Home Assistant for light entities like a Philips Hue light. The *hue* integration then uses this platform, in addition to context-specific code, to enable control over the light.
- **Device Integrations**: These allow communication between Home Assistant and third-party smart home devices, enabling them to be controlled through HA. For example, the *hue* integration allows Home Assistant to manage Philips Hue devices.
- **Automation Integrations**: These provide small automation logic components, with the *automation* integration being the most prominent. This integration allows users to create automations using configuration formats like YAML.

Through these extension mechanisms, Home Assistant can cater to the unique needs of every user, offering a continually expanding suite of device and automation options.

C. Device Integrations

This study focuses on device integrations, specifically the development of new integrations for previously unsupported devices or the addition of new features to existing device integrations. This area is the most active in terms of development, as 'building block' integrations- those that define core functionality- tend to remain relatively static, with few cases requiring entirely new domains for devices or functionalities. Additionally, in contrast to 'building block' and automation integrations, which are developed and maintained internally by the Home Assistant team, the majority of device integration code is contributed by third-party developers. Analyzing these integrations will provide valuable insights into the challenges faced by developers–particularly those who may not be fully familiar with the inner workings of Home Assistant–when attempting to integrate new devices or add new functionalities.

D. Third-Party Libraries

An important aside to consider is that Home Assistant integrations strictly do not include any protocol-specific code [12]. This means that when developing device integrations, code that interacts directly with a device's API must be placed in a standalone Python library and published to PyPI, remaining separate from the Home Assistant Core repository. These libraries typically consist of two main components: authentication mechanisms for issuing authenticated API requests, and data models that represent and provide functions to interact with the data. In device integrations, these libraries are imported as dependencies. This study will touch on the impact of these dependencies, and examine whether new contributors clearly understand this requirement, as a lack of awareness could lead to pull request rejections.

Listing 1: Example usage of a device library

```
#bad
status = requests.get(url("/status"))
#good
from phue import Bridge

bridge = Bridge(...)
status = bridge.status()
```

III. Related Work

A substantial body of research has been conducted on software practices in the IoT field; however, there is comparatively less focus specifically on smart home platform ecosystems, such as Home Assistant.

1) Smart Home Ecosystem: Zhou et al. [25] investigated various security vulnerabilities present in smart home devices across multiple platforms, identifying logic flaws related to authentication, faulty synchronization between devices and

IoT clouds, and more. Within Home Assistant's architecture, these responsibilities largely fall on device integrations, which manage tasks such as data retrieval, event handling, and device discovery on the network.

Anik et al. [18] conducted a recent study highlighting challenges faced by Home Assistant users when setting up automations. In Home Assistant, an automation is a rule that triggers an action when an event occurs and a (optionally) specified condition is met. Automations make use of the states, triggers, and services defined within integrations, and a variety of the identified challenges relate to incorrect interactions with this exposed behavior. This provides useful context to guide our study.

One of the most pertinent studies is by Wang et al. [24], who conducted an analysis of 330 bugs within Home Assistant device integrations, referred to as 'iBugs', developing a taxonomy of root causes. For instance, 45 of the identified iBugs were due to incomplete provision of a device's functionality within Home Assistant. The study provides a general model for device integration, emphasizing that development should prioritize four key areas: discovery, initialization, execution, and release. These insights provide much value in informing our analysis of the challenges involved in creating device integrations.

2) Comment Classification and Analysis: Turzo et al. [23] previously proposed a taxonomy for review comments using a manual classification approach. In contrast, we take a different approach by leveraging topic modeling to develop a more fine-grained taxonomy tailored specifically to the Home Assistant ecosystem.

While existing research has explored different facets of the Home Assistant ecosystem, no study to date has specifically focused on the challenges involved in the initial development and subsequent enhancement of device integrations. This study aims to bridge that gap by identifying common difficulties faced by developers, thereby providing insights that facilitate more rapid and more robust implementations of these integrations.

IV. Methodology

A. Data Collection and Preprocessing

To construct the initial dataset, pull requests from January 2021 to November 2024 were collected using an automated script that leveraged the GitHub REST API [3], yielding a total of 48,944 PRs. To bypass the API's rate limits, the script cycled through multiple GitHub Personal Access Tokens (PATs). The extracted metadata included creation and merge dates, the number of files changed, and lines of code changed.

Since this study focuses specifically on integration development, we isolated PRs that had the 'New integration' or 'New feature' checkboxes selected in the PR template, which is a mandatory requirement for all submissions to the Home Assistant Core repository. This information was extracted using BeautifulSoup [1], a Python library for HTML parsing. While label-based filtering (e.g., using 'integration: ') was

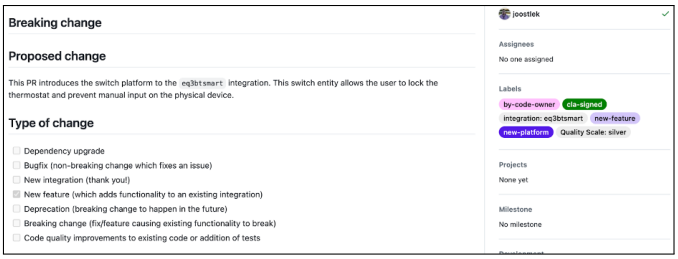

Figure 3: Home Assistant Core pull request template

considered, a preliminary analysis revealed that most PRs retrieved through this method were related to code quality improvements, bug fixes, and dependency updates—categories outside the scope of this study. The filtered dataset comprised 9,431 pull requests.

To supplement the collected metadata, comments from each pull request were scraped using two distinct GitHub API endpoints: 'review comments' (comments on code diffs, which can form threaded discussions) and 'issue comments' (standalone comments not tied to a specific code snippet). To preserve context for subsequent analysis, review comments were recursively grouped into their respective threads. Additionally, issue comments and review threads were interleaved chronologically based on timestamps, reconstructing the discussion as it appears in the GitHub UI. While this was not essential for the subsequent topic modeling, it proved valuable for manually referencing PR comments in their context.

B. Topic Modelling

To effectively address RQ1 and handle the large dataset, we employed BERTopic [19] for automated topic modeling. Prior to training the model, the scraped comment text was cleaned to reduce noise, including the removal of non-alphanumeric and non-punctuation characters, links, quotes, and code snippets. Pull requests that contained no comments or only empty comments after preprocessing were discarded, resulting in a final dataset of 5,091 PRs containing 22,865 threads.

For input to BERTopic, thread embeddings were generated using pre-trained sentence transformers. The model then applied dimensionality reduction via UMAP, followed by clustering using HDBScan. The hyperparameters for both algorithms are presented in Table I and II.

The dimensionality reduction step was configured with 15 neighbors to balance local and global structure, and cosine similarity, which is well-suited for high-dimensional data such as text-embeddings, was selected as the distance metric. Additionally, a `random_state` was set in order to mitigate the stochastic nature of the algorithm by default. For HDBSCAN, the most significant parameter choice is that of the minimum topic size; a value of 20 was observed to provide a strong balance between cluster coherence and filtering out of noise.

Topic representations were generated using CountVectorizer [2] with c-TF-IDF [19]. Finally, to fine-tune these representations based on the semantic relationship between keywords

Table I: UMAP hyperparameters

Parameter	Value
n_neighbors	15
n_components	5
min_dist	0.0
metric	cosine
random_state	42

Table II: HDBSCAN hyperparameters

Parameter	Value
min_cluster_size	20
min_samples	1
metric	euclidean
cluster_selection_method	eom
prediction_data	True

and the set of threads in each topic, KeyBERTInspired [10] was used. This eases subsequent grouping of similar topics.

Separate BERTopic models were trained for review threads and issue comments, as we hypothesized that their topics may differ. In particular , we suspected review comments to focus on code-level details, while issue comments would put more emphasis on addressing higher-level project requirements. For review comments, each of the pre-compiled threads from the previous stage was treated as a single unit when passed into the model. This approach yielded higher-quality clustering compared to processing individual comments, as the overall topic could be more effectively inferred. For issue comments, various topic segmentation algorithms were explored to group related comments before feeding them into the model. However, this approach proved ineffective due to the brevity of issue comments within a single pull request, making these algorithms unsuitable. Consequently, issue comments were passed into the model individually. Note that the discussion of results primarily focuses on review threads, as they were found to be the most insightful and subsumed the topics observed in issue comments. However, comparisons between the two will be made where relevant.

C. Threats to Validity

While we have taken measures to ensure the reliability and relevance of our findings, certain limitations and potential biases must be acknowledged.

1) Internal Validity: This study focuses exclusively on pull request comments, omitting other sources of discussion such as GitHub issues and community forums like Home Assistant Community [5]. As a result, certain challenges that do not surface in PR reviews may be overlooked. Additionally, the analyzed pull requests span approximately four years, raising the possibility that some identified topics are less relevant today, given the rapid evolution of Home Assistant. Where possible, we mitigated this by manually reviewing recent PRs to confirm the continued applicability of our findings. Finally, as a relatively small group of Home Assistant-employed maintainers conduct most reviews, their personal preferences could influence review patterns. However, we found no clear instances of this and recognize that some degree of subjectivity is inherent in code review.

2) External Validity: Our findings are based on the Home Assistant project, the largest open-source home automation platform. It is possible that these insights may not fully generalize to other closed and open-source projects, however we have observed that similar architectural patterns exist across all of the most popular platforms. For instance, OpenHAB's [7] "bindings" function similarly to Home Assistant's integrations, suggesting that some findings may transfer to other ecosystems.

3) Construct Validity: A significant number of review threads—over 30%—were classified as outliers by BERTopic, potentially leading to gaps in our taxonomy of code review discussions. Additionally, some large topics produced by the model due only to the recurrence of a common word like "entity" rather than any semantic similarity between threads were overly general and thus excluded. However, manual inspection of a sample of the outliers did not reveal missing categories; instead, most consisted of comments that were too brief or vague to be meaningfully clustered.

Therefore, despite these limitations, we believe our findings provide a representative overview of the key challenges developers face when integrating devices into Home Assistant.

V. Results

A. RQ1: What challenges do developers face when creating device integrations?

Through repeated rounds of refinement, the initial 189 topics were consolidated into a final set of 36 clusters, merging similar topics where possible. Manual analysis confirmed the topics to be of high quality, and outliers were discarded. These clusters are further organized into the taxonomy shown in Figure 4, comprising 7 high-level categories. The subsequent section explores the different topics, and expands on sub-topics in cases where we judge there to be valuable insights.

Code style and quality: This category encompasses discussions related to enforcing code style and ensuring adequate code quality through testing.

Code style/Typing and casting: Many of these threads reference guidelines outlined in Home Assistant's official style guide [6]. However, it is evident that developers are not always fully aware of these guidelines, or that the guide itself may not be comprehensive enough for less experienced contributors to meet the standards expected by project maintainers. The observed advice was wide ranging; an example is the frequent discussions, making up over 250 threads, regarding the use of constants. These often involve requests to replace magic numbers, commonly found as configuration variables, with constants and move them to the integration's `const.py`, or to utilize predefined values from Home Assistant's global `const.py`.

Imports: Discussions in this cluster centered on import path conventions. Within a component package/integration, imports should use relative paths, whereas imports from the

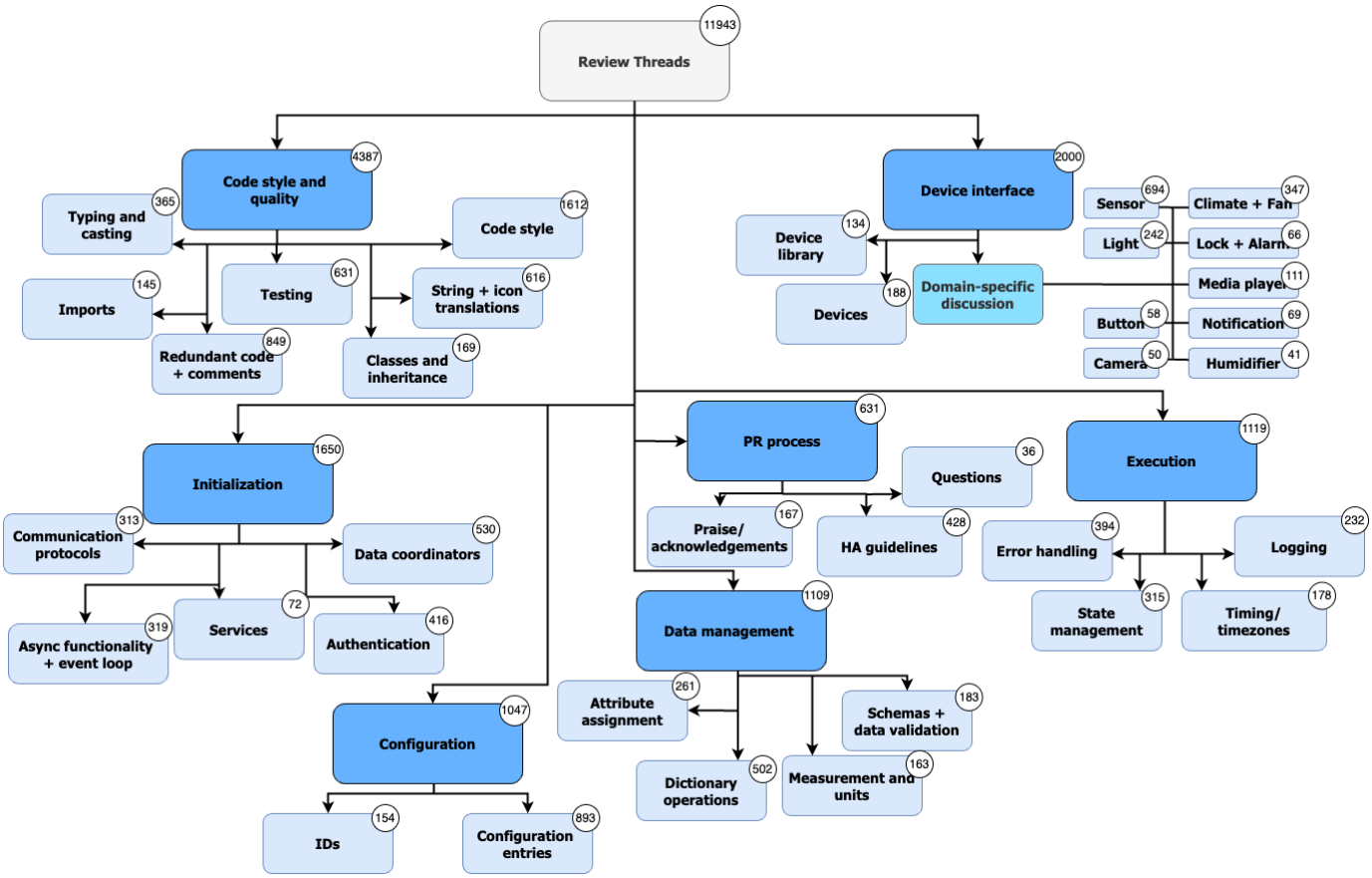

Figure 4: Taxonomy of review thread topics

broader Home Assistant project should be absolute. Additionally, maintainers emphasized that imports should be sorted alphabetically.

Classes and inheritance: Home Assistant employs an object-oriented architecture, for example with entity types inheriting from the `Entity` base class. A common issue identified by maintainers was developers unnecessarily reassigning attributes already initialized in base class constructors. Additionally, reviewers frequently requested that static initializations, which use identical values across all instances, be moved outside the constructor.

String + icon translations: Maintainers frequently advised developers to utilize the `strings.json` translation file within their integration to ensure entity and attribute names are presented in a user-friendly manner in the UI. For example, the following configuration handles a selector labeled "LED bar mode" in the UI, with its current state displayed accordingly:

Listing 2: Example strings.json entry

```
"entity": {
    "select": {
        "led_bar_mode": {
            "name": "LED bar mode",
            "state": {
                "off": "Off",
                "co2": "Carbon dioxide",
                "pm": "Particulate matter"
            }
        }
    }
}
```

A similar approach is recommended for icon translations using `icons.json`, which allows different states to be represented with appropriate UI icons:

Listing 3: Example icons.json entry

```
...
"state": {
  "new_moon": "mdi:moon-new",
  "first_quarter": "mdi:moon-first-quarter",
  "full_moon": "mdi:moon-full",
  "last_quarter": "mdi:moon-last-quarter"
}
```

Testing: Prior studies have highlighted a general lack of rigorous testing within the IoT ecosystem [20, 21] . Many contributors initially fail to provide sufficient test coverage to meet project requirements. A common example is the need for 100% test coverage of config flows, which maintainers frequently had to point out during reviews. Beyond coverage requirements, maintainers often suggested best practices to improve testing quality and maintainability:

– Contributors are encouraged to use pytest fixtures to create consistent test environments, which should be defined

in `conftest.py`. Additionally, when testing multiple scenarios or variations of input, maintainers frequently recommend using `pytest.mark.parametrize()` to efficiently handle a large number of test cases.
- In certain cases, contributors were pointed towards snapshot testing, which is a relatively new addition to Home Assistant's testing strategy and is particularly useful for verifying large outputs, such as dumps of states or entity registry items. This technique ensures that data structures remain consistent across code changes.
- Reviewers emphasized that tests should patch the external device library rather than patching Home Assistant's own code. This approach ensures that Home Assistant's integration logic is fully covered while avoiding interference from the device library's functionality, whose testing is outside the project's scope.

Initialization: Initialization encompasses integrations' various responsibilities related to authenticating the device, setting up polling, and registering functionality with HA Core. This has previously been identified as a rich area for bugs to develop [24, 20], leading it to be a clear area of concern for maintainers.

Authentication: Many physical devices require Home Assistant to establish an authenticated connection before allowing control. During the configuration process, users enter their credentials, which are then used to authenticate via the device's third-party library.

Listing 4: Example authentication flow

```
async def async_step_user(
    self, user_input: dict[str, Any] | None =
        None
) -> ConfigFlowResult:
    """Handle the cloud logon step."""
    ...
    session = async_get_clientsession(self.
        hass)
    api = AquacellApi(session, user_input[
        CONF_BRAND])
    try:
        refresh_token = await
            api.authenticate(
            user_input[CONF_EMAIL], user_input
                [CONF_PASSWORD]
        )
    ... # error handling
```

Reviewers frequently highlight inadequate handling of authentication failures. However, the most commonly raised issue is the absence of reauthentication mechanisms. When a user changes the password of a linked device, the integration should detect this and prompt the user to reauthenticate, rather than requiring them to manually delete and recreate the configuration entry.

Data coordinators: Home Assistant provides the `DataUpdateCoordinator` class to manage API polling within a device integration, ensuring efficient scheduling within the event loop. However, many developers were unaware of this built-in functionality and implemented their own polling mechanisms. A common mistake highlighted by maintainers was allowing users to configure the polling interval in the UI; this is not permitted in Home Assistant, and the integration author is expected to define an appropriate default value. Additionally, some developers proposed polling intervals shorter than five seconds, which is explicitly disallowed by the project.

Services[1]: Integrations can define services to control entities within a component or trigger external scripts, with their descriptions stored in the integration's `services.yaml` file. Several review threads addressed issues with services not being correctly registered using `hass.services.register(...)`- for example, only registering them conditionally, which can cause unexpected errors in automations. Additionally, some developers initially exposed functionality that should not be available to users, such as initialization tasks.

Async functionality + event loop: Home Assistant restricts access to core API objects to a special thread called the **event loop**, where components schedule their tasks for execution. Tasks that perform blocking I/O, such as fetching new data from a device, can suspend while waiting for a response. Maintainers frequently pointed out common mistakes when working with asynchronous functionality:

- Since synchronous calls block execution, they should be offloaded to a separate thread using the executor loop. In Home Assistant code, this is done with `hass.async_add_executor_job(...)`, with calls of functions within library code performed using `loop.run_in_executor(...)`.
- When grouping multiple tasks, developers should use `asyncio.gather()` to run them concurrently rather than sequentially. A common mistake is awaiting each task individually in a loop, as shown in Listing 5:

Listing 5: Example use of asyncio.gather()

```
async def _async_update_data(self):
    # bad
    for bed in self.client.beds.values():
        await bed.fetch_pause_mode()

    # good
    await asyncio.gather(
        *[bed.fetch_pause_mode() for bed
            in self.client.beds.values()]
    )
```

- Normal functions that do not perform I/O or call coroutines should be marked with the `@callback` annotation, marking them as safe to run in the event loop because they do not suspend execution. While they can schedule new tasks, they do not await results, making a common use case for them listeners for events or service action calls.

[1]Services have recently been renamed to actions within the Home Assistant ecosystem

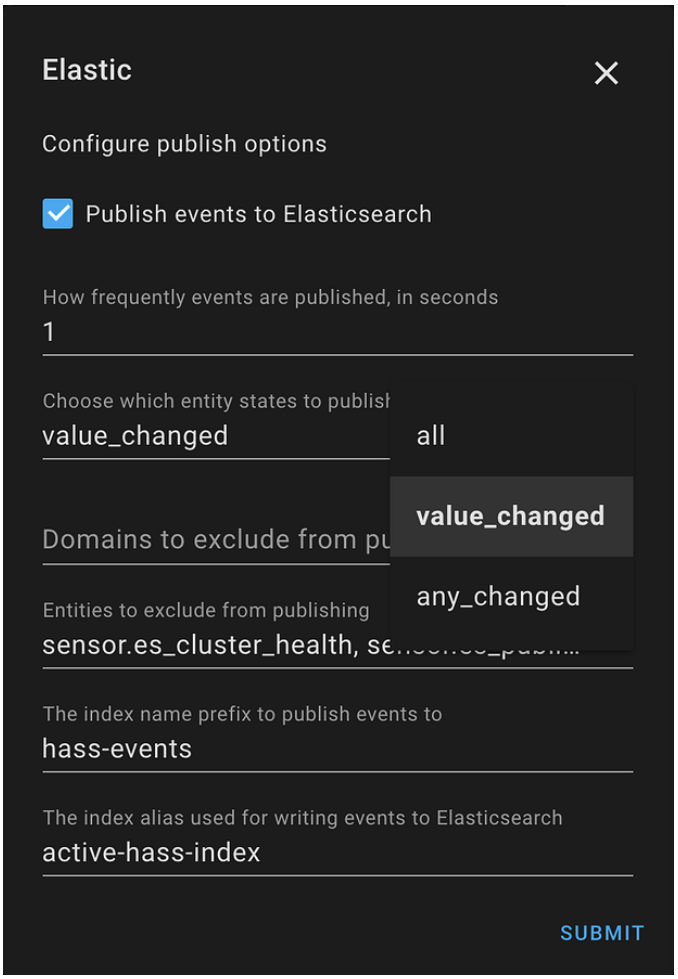

Figure 5: Sample config flow [4]

Communication protocols: This category includes discussions on establishing connections using protocols beyond HTTP, particularly MQTT, Bluetooth Low Energy (BLE), and Home Assistant's WebSocket API. Many review threads were highly specific to individual use cases, making it difficult to extract generalizable best practices, and suggesting that the integration of these protocols often follows an ad-hoc approach. A recurring challenge was the difficulty of testing, highlighting the need for more standardized testing practices when working with these communication protocols.

Configuration: The configuration topic deals primarily with writing the code for **configuration flows**- displays in the UI through which users set up integrations (Figure 5). While configuration was previously managed in Home Assistant through manual manipulation of a `configuration.yaml` file, this is now forbidden, and instead must be performed in the UI. The integration's `config_flow.py` file controls how data is presented and processed within this setup flow. Additionally, **options flows** can be used to expose optional settings that allow users to tweak integration behavior after the initial setup. The following common mistakes were observed:

Configuration entries:

– Many developers did not correctly handle the unloading of config entries when users removed or disabled an integration. Unloading should clean up all associated resources, such as closing connections and removing entities from Home Assistant's entity registry.

Listing 6: Example unloading of a config entry

```
async def async_unload_entry(hass:
    HomeAssistant, entry: ConfigEntry) ->
    bool:
    ...
    unload_ok = await hass.config_entries.
        async_unload_platforms(entry,
        PLATFORMS)
    if unload_ok:
        hass.data[DOMAIN].pop(entry.
            entry_id)

    await async_unload_services(hass)

    return unload_ok
```

Reviewers often directed developers to use the `entry.async_on_unload(...)` listener, which serves a similar purpose.

– When registering entities during the config setup process, `async_add_entities()` should be called only once, passing in all entities. Some developers mistakenly called it multiple times, leading to inefficiencies and potential issues.

– Developers often misused `async_setup()` when implementing config flows. In integrations that use UI-based configuration (rather than YAML), `async_setup_entry()` should be used exclusively, and reviewers had to request the removal of `async_setup()`.

– Developers often stored configuration parameters (such as a device's host and port) under arbitrary key names within the config flow object. Home Assistant follows a naming convention where predefined constants, such as `CONF_PORT` from `const.py`, should be used to ensure consistency across integrations.

– A recurring issue was developers updating YAML configuration schemas, such as adding new possible values, despite YAML-based configuration no longer being allowed for new integrations. Maintainers frequently reminded contributors that all configuration must now be handled through the UI using config flows.

IDs: To prevent users from accidentally setting up the same device multiple times in the UI, each config flow should be assigned a unique ID. This ensures that Home Assistant can recognize and prevent duplicate entries for the same device. There are a helpers available for this, as demonstrated in Listing 7- the `_abort_if_unique_id_configured()` being used to terminate the configuration flow if an entry already exists.

Listing 7: Handling unique IDs inside a config flow step

```
async def async_step_user(
    self, user_input: dict[str, Any] | None =
        None
) -> ConfigFlowResult:
    errors: dict[str, str] = {}
    if user_input:
        client = MyClient(user_input[CONF_HOST
            ])
        try:
            identifier = await client.
                get_identifier()
        ...
        else:
            await self.async_set_unique_id (
                identifier)
            self.
                _abort_if_unique_id_configured()

            return self.async_create_entry(
                title="MyIntegration",
                data=user_input,
            )
    ...
```

Some contributors forgot to assign a unique ID, or used values that can change over time, such as IP addresses. When available, maintainers recommended the use of device serial numbers or other stable hardware identifiers; note that the ID must only be unique within the integration, not globally.

Data Management: This cluster contained threads around how integrations store, manipulate, and validate data within Home Assistant.

Attribute Assignment: Entities in Home Assistant can have a variety of attributes, defined using Python's `@property` decorator. However, for attributes that remain constant rather than being dynamically computed each time they are accessed, reviewers frequently recommended using the shorthand notation of assigning values directly to instance variables. The most common recommendation was to set `device_info` in this way:

Listing 8: Example setting of property attributes

```
def __init__(self, device: KaleidescapeDevice)
     -> None:
    # preferred
    self._attr_device_info = DeviceInfo(...)

# discouraged
@property
def device_info(self) -> DeviceInfo:
    return DeviceInfo(...)
```

Reviewers also commonly advised converting instance attributes to class attributes when values remain static across all instances e.g. `self._attr_state_class = STATE_CLASS_MEASUREMENT`.

Dictionary Operations: Much of the data access in Home Assistant relies on dictionaries. This was one of the largest clusters, but similar to some other clusters, there were few extractable common themes; most threads were highly use-case specific. One common piece of advice was not to use `.get()` in cases where an attribute is guaranteed not to be `None`, as it can obscure expectations about which attributes are always required and may hide errors when a required key is missing. In these cases, direct dictionary access is preferred. Similarly, for large dictionaries, maintainers recommended replacement with dataclasses to improve typing and readability.

Measurements and units: Accurate handling of measurements and units is essential in Home Assistant to ensure that sensor readings are correctly interpreted, automations trigger as expected, and data is displayed properly in the UI. Developers often forgot to include the `native_unit_of_measurement` field of entity descriptions, having to be prompted by reviewers.

Schemas and data validation: Home Assistant relies on the Voluptuous [8] library for validating incoming YAML. While YAML-based configuration has been replaced by UI flows as mentioned previously, YAML is still generated in the background for input-driven features such as config flows and service actions.

Listing 9: Voluptuous schema validators

```
OPTIONS_SCHEMA = vol.Schema(
    {
        vol.REQUIRED("field_name", default="
            default_value"): str,
    }
)

class ExampleOptionsFlow(config_entries.
    OptionsFlow):
    async def async_step_init(
        self, user_input: dict[str, Any] |
            None = None
    ) -> FlowResult:
        return self.async_show_form(
        data_schema = self.
            add_suggested_values_to_schema(
            OPTIONS_SCHEMA, self.entry.options
        )
    )
```

Many contributors manually checked whether required fields were present in the config entry dictionary, even though Voluptuous schemas already enforce this, leading to unnecessary conditional logic in initialization code. Reviewers also frequently directed developers to the `add_suggested_values_to_schema()` helper function, which can be used to pre-fill previously entered values instead of resetting to defaults when presenting a reconfiguration flow.

PR Process: Comments in this cluster included general praise and acknowledgements, questions on implementation, and, most valuable to developers, advice for conforming to the rules of the Home Assistant project. These are elaborated on in the following section.

Home Assistant guidelines: As with many open-source projects, Home Assistant enforces specific guidelines to stan-

dardize the review process and maintain code quality. However, as evidenced by multiple review threads, contributors do not always adhere to these expectations.

- Many developers mistakenly included dependency bumps within their feature or bugfix pull requests. Home Assistant requires that these updates be handled in dedicated preliminary or follow-up PRs.
- To keep reviews focused, pull requests should be limited to a single platform. Recall that a platform describes the use of an entity integration (a.k.a building block integration, such as *light*) for common functionality; if a device has multiple types of entities, changes should be made to only one per pull request.
- Contributors must ensure that their PRs remain in sync with the latest `dev` branch, necessitating rebases if the timeline of an integration is long. Reviewers commonly had to catch unintended code changes that slip in during this process.

Execution: The execution cluster covered issues concerning the handling of errors during runtime, logging, and timeouts, for example on polling.

Error handling: Since Home Assistant integrations communicate with external devices and accept user configurations, errors are inevitable. Code must handle these errors robustly and provide clear feedback to users. Reviewers identified several common mistakes:

- Broad exceptions should not be caught; outside of config flows, `except Exception` is not permitted. Developers should only catch specific errors raised by the device library (which are imported), ensuring better user messaging and handling.
- Logging errors alone is insufficient, and users should use Home Assistant's provided error types to raise exceptions to be handled by the HA Core: `ServiceValidationError` when the user provides incorrect input, and `HomeAssistantError` for other failures, such as communication issues with the device.
- Error messages should be informative to help users troubleshoot issues; contributors frequently did not meet this requirement. Additionally, errors should support translations via the `strings.json` file.

Listing 10: Example error translation

```
# strings.json
{
  "exceptions": {
    "invalid_index": {
      "message": "Invalid index selected,
          expected: {expected}, got
          {index}"
    }
  }
}

# service action call
async def async_select_index(hass:
    HomeAssistant, index: int) -> None:
    """Setup the config entry for my
        device."""
    try:
        check_index(index)
    except ValueError as exc:
        raise ServiceValidationError(
            translation_domain=DOMAIN,
            translation_key="invalid_index
                ",
            translation_placeholders={
                "index": index,
                "expected": expected,
            },
        ) from exc
```

- Developers often included unnecessary code inside try blocks. Home Assistant enforces that try blocks contain only code that may raise exceptions to improve clarity and prevent unintended error suppression.

Logging: Effective logging is essential for both users diagnosing issues and developers debugging integrations. Reviewers pointed out common mistakes, for example developers both logged errors and raised a `HomeAssistantError`; this resulted in duplicated logs, as raising the exception automatically logs it, and additional messaging can be provided as an argument. As well as this, developers often logged messages at the `info` level instead of `debug`. The info level should be reserved for messages relevant to users, while debug should be used for developer-focused diagnostics.

Timing & timezones: Many developers retrieved timestamps from a device's API without considering whether the values were in local time or UTC. Home Assistant uses UTC internally and expects all integrations to follow this convention to ensure consistency across automations, logging, and UI displays; maintainers recommended using the `homeassistant.util.dt.utcnow()` helper for this. Additionally, developers often attempted to handle request timeouts directly within integration code. However, maintainers emphasized that timeouts should be implemented and managed within the device library instead.

State management: Home Assistant maintains an internal state machine where each entity has a single state (e.g., "on" or "off" for a light) and additional attributes (e.g., brightness, color). Similar to some other clusters, many threads were use-case specific. A common mistake developers made was redundantly calling `async_write_ha_state()` after modifying entity attributes. Home Assistant automatically detects attribute changes and updates the state, making explicit calls unnecessary in most cases- it is only required if state changes occur outside the normal update cycle, such as when an integration receives push updates from an external API

Listing 11: Example async_write_ha_state() usage

```
async def async_turn_off(self, **kwargs):
    """Turn the switch off."""
    await self._api_client.turn_off()
    self._attr_is_on = False
```

```
    self.async_write_ha_state() # unnecessary

async def external_update_handler(new_state:
    bool):
    """Handle an external state change (e.g.,
        push update from API)."""
    my_switch_entity._attr_is_on = new_state
    my_switch_entity.async_write_ha_state()
        # necessary
```

Integration code specifics: This cluster discusses how Home Assistant interfaces with various types of components and devices. Included are clusters focusing on many of the principle entity types, for example sensors and cameras.

Device library: As previously mentioned, all protocol specific code with the device should be encapsulated in a separate library and published on PyPI, rather than handled within the Home Assistant integration. Despite this guideline, reviewers frequently had to remind contributors to move certain logic, such as request handling, authentication, and data parsing, out of the integration and into the device library. With this guideline, the project tries to maintain and a clear separation of concerns.

Devices: In Home Assistant, devices serve as the logical grouping for entities. During code reviews, reviewers highlighted two common mistakes developers make when working with devices:

- Many developers underutilized the `DeviceInfo()` helper when assigning `self._attr_device_info`. This helper provides metadata such as name and manufacturer related to devices, ensuring they are properly registered and represented.
- Reviewers encouraged the specification of a device class for each entity, which is used to reflect the type of data it represents. For example, sensor entities can use `SensorDeviceClass.CARBON_DIOXIDE`. Setting a device class automatically displays the correct icons in the UI and provides some type safety by restricting the possible units of measurement selected.

Domain-specific discussion: Nine clusters relating to the different entity types formed their own cluster; one can hypothesize that these are the most common entity types, in particular sensor and binary sensor entities, which formed one of the largest overall clusters. The review threads were highly use-case specific, indicating that working with the details of individual entity types present unique integration challenges.

B. RQ2: What are the characteristics of device integration pull requests?

To answer this research question, we analyze the various aspects of pull requests related to device integrations in the Home Assistant repository.

1) What topics appear most frequently in hard-to-merge cases?: We define ”hard-to-merge” pull requests as those in the upper quartile of merge times, reflecting the additional effort required for approval, and likely involving multiple

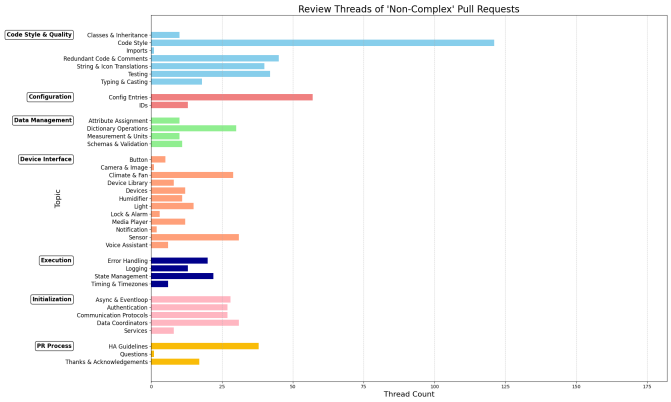

Figure 6: Review thread topics of ’non-complex’ PRs

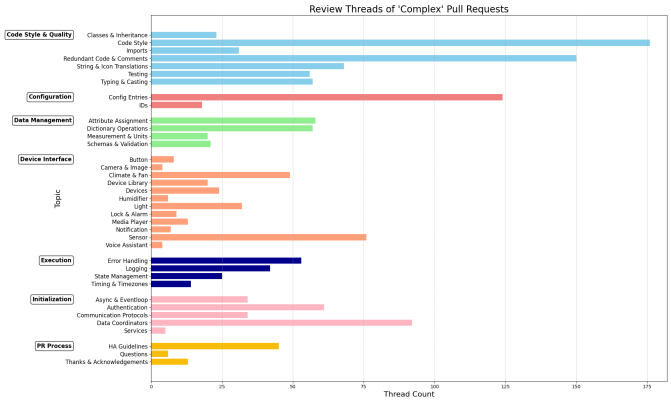

Figure 7: Review thread topics of ’complex’ PRs

review iterations. To analyze review discussions, we sample 400 of these pull requests along with 400 ”non-complex” pull requests from the remaining set. The classification of their review threads is presented in Figures 6 and 7.

While longer merge times naturally correlate with increased comment volume, our analysis reveals that certain technical areas disproportionately contribute to pull request complexity. For example, discussions around authentication and data coordinators both more than double in size, highlighting that they can be challenging areas to resolve. This complexity may stem from some of the concerns discussed previously, including difficulty in managing asynchronous data fetching and reauthentication mechanisms.

Error handling also sees a substantial rise in review discussions, suggesting that developers frequently struggle with handling edge cases and error recovery mechanisms. Given that Home Assistant integrates with a diverse range of third-party APIs and hardware devices, dealing with unexpected failures is a persistent challenge which proves difficult for contributors to deal with.

Among the entity types, sensor-related discussions exhibit the largest distribution increase. This could merely be attributed to the prevalence of sensor devices in the smart home ecosystem, or could be related to their inherent complexity, which frequently involves real-time data streaming, varying

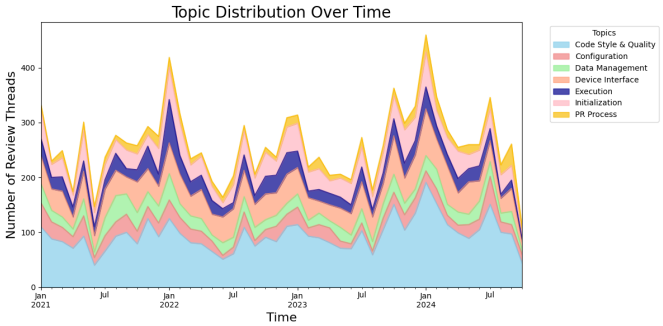

Figure 8: Topic distribution over time

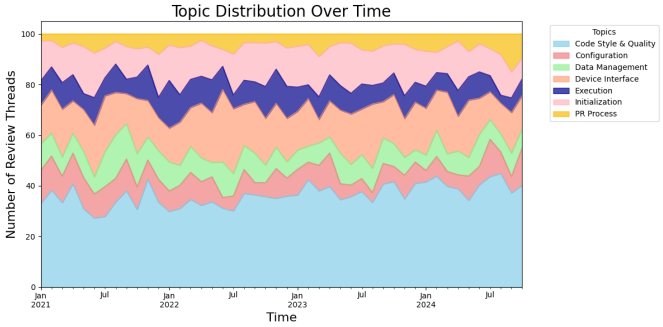

Figure 9: Normalized topic distribution over time

communication protocols (Zigbee, Z-wave, ...) and unit conversions.

A notable increase is observed in code style and redundant code discussions, which implies that contributors often require multiple review iterations to align their implementations with Home Assistant's best practices. This suggests a potential gap in either documentation clarity and comprehensiveness, which could be mitigated by refining the provided linting configuration to catch common issues more effectively.

Finding: *The strongest predictors of PR merge times appear to be code style issues, configuration flows, coordinators, error handling, and authentication. This shows a clear emphasis on working with the overall architecture of Home Assistant rather than device implementation details.*

2) How Does Topic Frequency Change Over Time?: We examine the distribution of topics in pull request review threads throughout the data collection period, spanning from January 2021 to October 2024. We analyze both the absolute numbers and the relative percentage shares of each high-level category. The findings are illustrated in Figures 8 and 9.

The analysis reveals that review threads focused on code style and quality consistently represent the largest category, accounting for approximately 30-40% of the total. This is followed by the device interface and initialization categories, which each contribute around 15-20%. Notably, the distribution of review threads has remained relatively stable over the past four years, suggesting that developers continue to

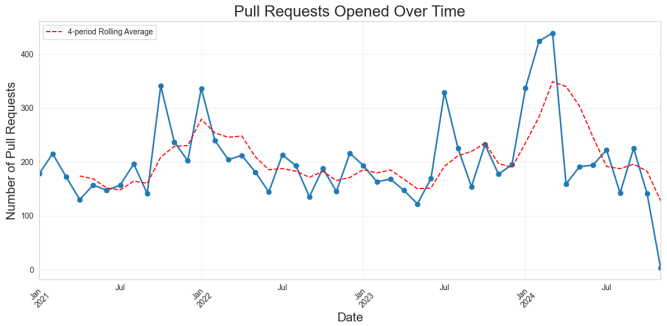

Figure 10: PRs opened over time

encounter similar issues, necessitating repeated feedback from reviewers. This trend indicates a potential stagnation in the collective knowledge base among contributors.

In absolute terms, the overall number of review threads has not increased significantly throughout the observed period, remaining relatively constant. This is reflected in the number of pull requests opened over the same period seen in Figure 10. Interestingly, there is a discernible pattern of peak reviewer activity in January each year, along with noticeable spikes during the July-August period. These trends may be attributed to contributors acquiring new home automation hardware during the Christmas season, as well as as the tendency to start new projects during the summer months.

Finding: *The distribution of topics discussed in review threads within the Home Assistant project has remained relatively stable over time, suggesting that developers are facing the same challenges.*

3) How do pull requests differ between codeowners and non-codeowners?: Home Assistant leverages GitHub's CODEOWNERS feature, allowing the designation of individuals or teams as responsible for specific parts of the codebase. As part of this, code owners are automatically requested for review when a pull request is opened for their respective code section.

As illustrated in the left portion of Figure 11, there is a two-to-one ratio of pull requests submitted to an integration by a contributor who is not the codeowner compared to those opened by the codeowner themselves. However, when we broaden our definition to consider *any* codeowner listed in the `CODEOWNERS.txt` file, regardless of the involved integration, over 75% of pull requests are initiated by codeowners.

This indicates that the majority of pull requests are contributed by individuals who are actively engaged with the project, rather than by one-off contributors. This could suggest a reliance on a small group of individuals, most of whom are volunteers, for the overall health of the project. It could also discourage new contributors, as the expectation that most code changes come from experienced contributors may create a perceived barrier to entry. We can see from Figure 11 that

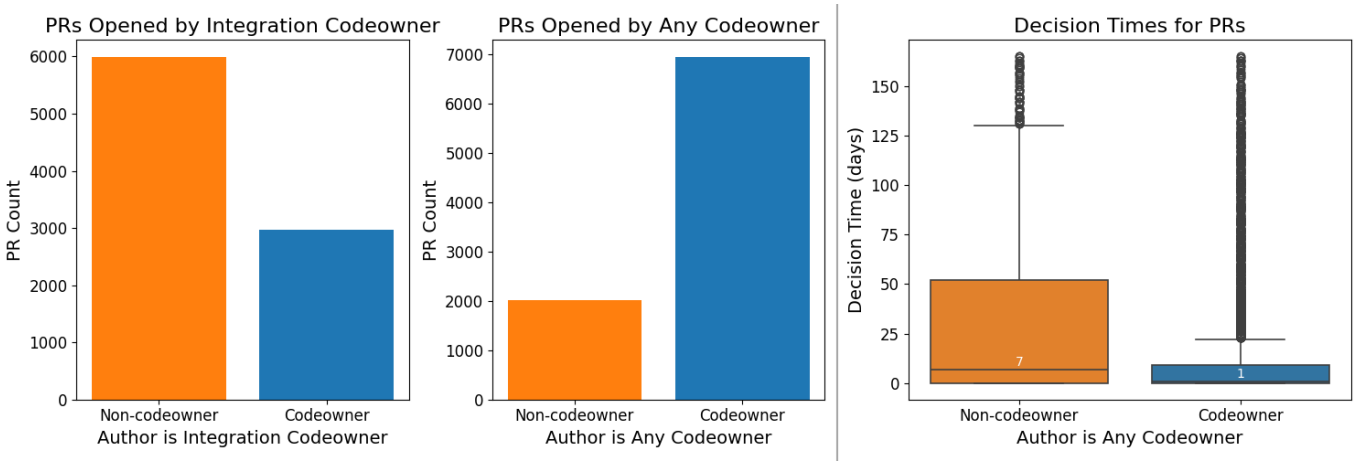

Figure 11: Codeowner vs Non-codeowner pull requests

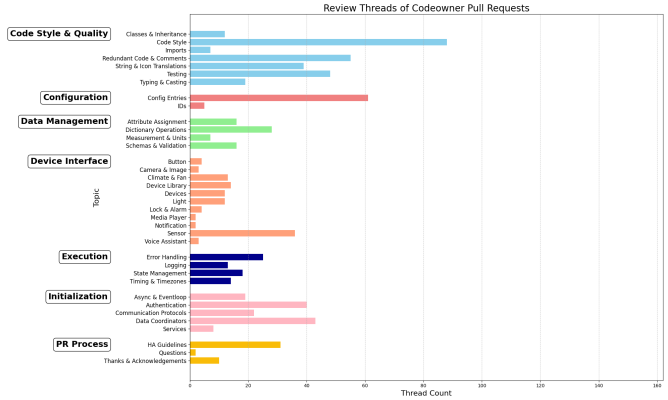

Figure 12: Review thread topics in PRs opened by codeowners

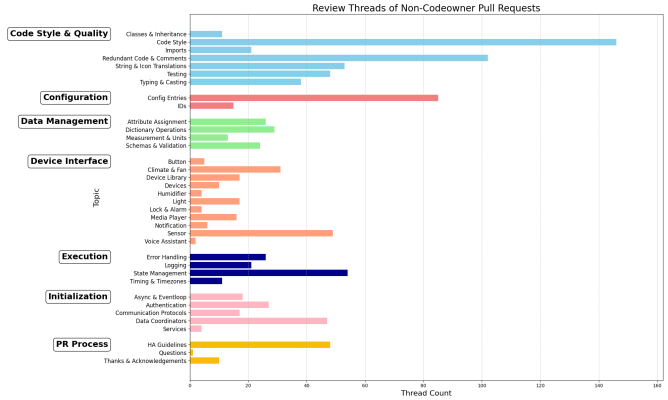

Figure 13: Review thread topics in PRs opened by non-codeowners

non-codeowners' PRs have a median decision time of 7 days compared to the 1 day of contributors more familiar with the project.

Using this broader definition of a codeowner, we now analyze the differences in topic distribution between review threads in pull requests opened by codeowners and non-codeowners. The results are presented in Figures 12 and 13.

Compared to non-codeowners, PRs opened by codeowners show a significant reduction in review threads related to code style and quality, as well as discussions on HA guidelines. This suggests that domain expertise plays a key role in meeting maintainers' coding standards. Additionally, there is a noticeable decrease in configuration-related discussions, previously identified as a common challenge for contributors. However, other difficult areas, such as managing data coordinators and error handling, do not exhibit a similar decline, indicating that these aspects remain complex even for experienced developers. This highlights an opportunity to improve documentation and knowledge-sharing efforts in these areas.

Finding: *The large majority of pull requests are opened by developers who actively engage with the project as codeowners. While their work undergoes less scrutiny regarding code style and project guidelines, they still encounter many of the same challenges as non-codeowners, particularly in complex areas such as error handling and data fetching.*

4) What pull request characteristics lead to longer merge times?: Using metadata from approximately 9,000 pull requests, we analyze the relationship between various characteristics and decision times (whether merged or closed). To mitigate the influence of outliers, we apply a 98th percentile cutoff for each feature. Additionally, we use a log scale for improved visualization. Pull requests that were closed (˜2,000) are highlighted in red, while merged PRs (˜7,000) are shown in blue. The results are presented in Figure 14.

The majority of pull requests are processed within the first three days, suggesting an efficient review process within the project.

The boxplots reveal a clear trend: as the number of files and lines of code (LOC) changed increases, the IQR and median decision time both rise. Specifically, the median number of files changed grows from 3 in the 0–3 day range to 5 in the 30+ day range, indicating that more complex PRs (with more

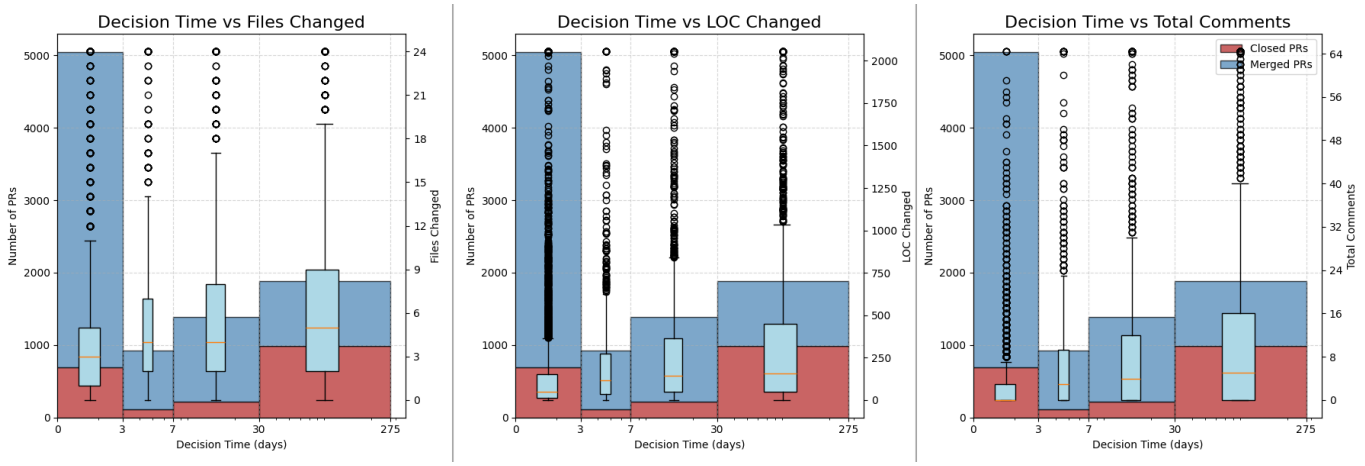

Figure 14: Relationship between PR characteristics and decision times

modified files) tend to require longer review periods, while the median for LOC changed grows from 50 to 160. As a consequence, developers may be able to improve their chances of faster merges by limiting the scope of changes within a single PR. However, this is not a strict rule, as some outliers with a high number of code and file changes are still processed relatively quickly.

We also observe a correlation between the total number of comments on a PR and the decision time. Notably, many PRs that involve simple fixes can be merged without any feedback, having a median of 0 in the largest bin of 0-3 days. However, once changes are requested, PRs frequently exceed the three-day mark before undergoing re-review and reaching a decision. This suggests that the re-review process may not be particularly swift. During our manual examination of the comments, we encountered numerous instances where contributors experienced lengthy delays awaiting re-review. In some cases, contributors resorted to pinging reviewers in the comments, which is against the repository's guidelines.

A notable proportion of closed PRs fall within the 0–3 day time bin, indicating a swift screening process where maintainers quickly reject unsuitable contributions. Interestingly, the highest concentration of closed PRs occurs beyond the 30-day mark. Our analysis of review discussions suggests that many of these cases involve PRs undergoing multiple revision cycles, leading to developers feeling overwhelmed and abandoning the PR. This highlights a potential challenge: prolonged review iterations can discourage contributors, especially first-time or inexperienced contributors, leading to an increased likelihood of PRs being left incomplete and closed.

Finding: *There is a clear relationship between the complexity (files, lines of code changed) of PRs and length of review periods. While many simple PRs can be merged swiftly, those requiring changes often face significant delays, which may deter contributors. The prevalence of closed PRs within both the 0–3 day and 30+ day time bins suggests a quick rejection of unsuitable contributions on one hand, and challenges in managing lengthy revision cycles on the other.*

VI. Conclusion

In this paper, we conducted the first empirical study of review threads within the smart-home ecosystem, developing a taxonomy of seven high-level topics that highlight the diverse challenges developers encounter. Based on this, we identified representative examples of repeated mistakes made within each topic by developers, along with what topics frequently appear in hard-to-merge pull requests. We examined the factors contributing to prolonged merge times for device integration pull requests, aiming to provide insights that could inform best practices for accelerating the review process. Looking ahead, we believe future research could leverage our findings from RQ1 to create a code-analysis tool, fine-tuned from a large pre-trained model, to serve as a knowledge base for enabling new contributors to quickly grasp Home Assistant's integration requirements and standards.

References

[1] BeautifulSoup. Retrieved March 1, 2025. URL: https://pypi.org/project/beautifulsoup4/.

[2] CountVectorizer, scikit-learn. Retrieved March 3, 2025. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.

[3] Github REST API. Retrieved March 1, 2025. URL: https://docs.github.com/en/rest?apiVersion=2022-11-28.

[4] HAC Config Flow Thread. Retrieved March 3, 2025. URL: https://community.home-assistant.io/t/customize-

display-value-of-combobox-items-for-integration-config-flow/284872.

[5] Home Assistant Community. Retrieved March 2, 2025. URL: https://community.home-assistant.io/.

[6] Home Assistant Style Guide. Retrieved March 3, 2025. URL: https://developers.home-assistant.io/docs/development_guidelines/.

[7] OpenHAB. Retrieved March 3, 2025. URL: https://www.openhab.org/.

[8] Voluptuous. Retrieved March 3, 2025. URL: https://pypi.org/project/voluptuous/.

[9] Github Octoverse, 2024. Retrieved March 1, 2025. URL: https://github.blog/news-insights/octoverse/octoverse-2024/#the-state-of-open-source.

[10] KeyBERTInspired, 2024. Retrieved March 3, 2025. URL: https://maartengr.github.io/BERTopic/api/representation/keybert.html.

[11] Apple HomeKit, 2025. Retrieved March 1, 2025. URL: https://www.apple.com/home-app/.

[12] Building a Python library for an API, 2025. Retrieved March 1, 2025. URL: https://developers.home-assistant.io/docs/api_lib_index/.

[13] Google Home, 2025. Retrieved March 1, 2025. URL: https://home.google.com/welcome/.

[14] Home Assistant, 2025. Retrieved March 1, 2025. URL: https://www.home-assistant.io/.

[15] Home Assistant Core, 2025. Retrieved March 1, 2025. URL: https://developers.home-assistant.io/docs/architecture/core.

[16] Integration types, 2025. Retrieved March 1, 2025. URL: https://developers.home-assistant.io/docs/architecture_components.

[17] Samsung SmartThings, 2025. Retrieved March 1, 2025. URL: https://www.smartthings.com/.

[18] Sheik Murad Hassan Anik, Xinghua Gao, Hao Zhong, Xiaoyin Wang, and Na Meng. Automation configuration in smart home systems: Challenges and opportunities, 2024. URL: https://arxiv.org/abs/2408.04755, https://arxiv.org/abs/2408.04755 arXiv:2408.04755.

[19] Maarten Grootendorst. Bertopic: Neural topic modeling with a class-based tf-idf procedure. *arXiv preprint arXiv:2203.05794*, 2022.

[20] Amir Makhshari and Ali Mesbah. Iot bugs and development challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 460–472, 2021. https://doi.org/10.1109/ICSE43902.2021.00051 doi:10.1109/ICSE43902.2021.00051.

[21] Pedro Martins Pontes, Bruno Lima, and João Pascoal Faria. Test patterns for iot. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 63–66, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3278186.3278196 doi:10.1145/3278186.3278196.

[22] Statista. Smart home - worldwide market outlook, 2024. Retrieved March 1, 2025. URL: https://www.statista.com/outlook/cmo/smart-home/worldwide.

[23] Asif Kamal Turzo and Amiangshu Bosu. What makes a code review useful to opendev developers? an empirical investigation, 2023. URL: https://arxiv.org/abs/2302.11686, https://arxiv.org/abs/2302.11686 arXiv:2302.11686.

[24] Tao Wang, Kangkang Zhang, Wei Chen, Wensheng Dou, Jiaxin Zhu, Jun Wei, and Tao Huang. Understanding device integration bugs in smart home system. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 429–441, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3533767.3534365 doi:10.1145/3533767.3534365.

[25] Wei Zhou, Chen Cao, Dongdong Huo, Kai Cheng, Lan Zhang, Le Guan, Tao Liu, Yan Jia, Yaowen Zheng, Yuqing Zhang, Limin Sun, Yazhe Wang, and Peng Liu. Reviewing iot security via logic bugs in iot platforms and systems. *IEEE Internet of Things Journal*, 8(14):11621–11639, 2021. https://doi.org/10.1109/JIOT.2021.3059457 doi:10.1109/JIOT.2021.3059457.