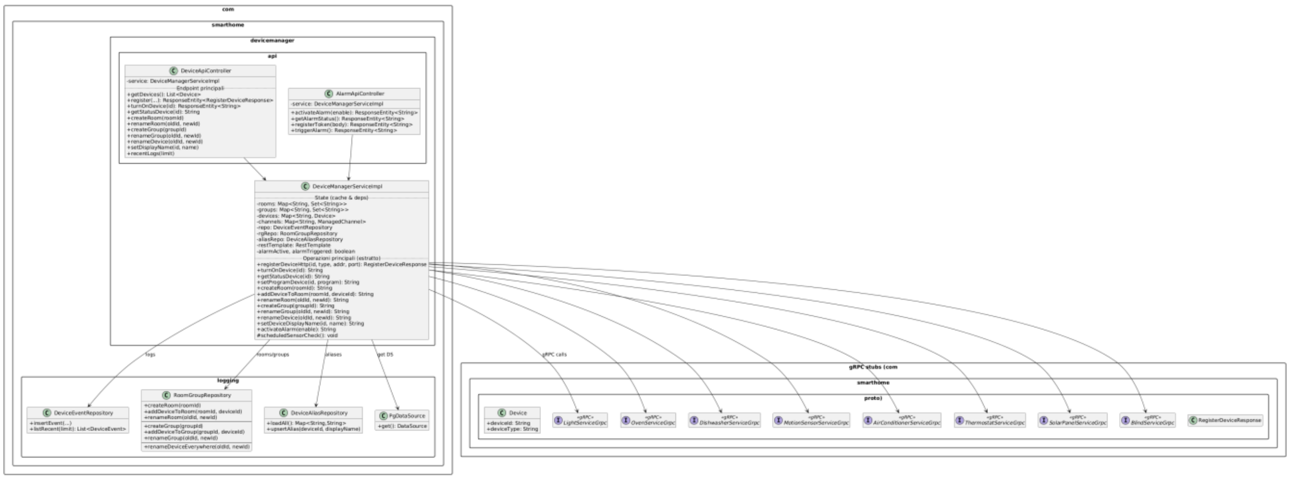
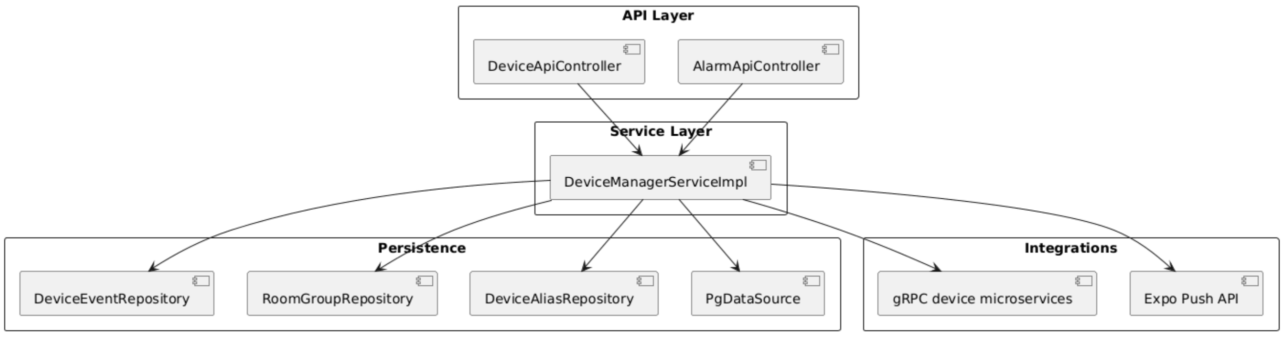
Project Structure


com
smarthome
devicemanager
api
DeviceApiController
-service: DeviceManagerServiceImpl
Endpoint principali
+getDevices(): List<Device>
+register(...): ResponseEntity<RegisterDeviceResponse>
+turnOnDevice(id): ResponseEntity<String>
+getStatusDevice(id): String
+createRoom(roomId)
+renameRoom(oldId, newId)
+createGroup(groupId)
+renameGroup(oldId, newId)
+renameDevice(oldId, newId)
+setDisplayName(id, name)
+recentLogs(limit)
AlarmApiController
-service: DeviceManagerServiceImpl
+activateAlarm(enable): ResponseEntity<String>
+getAlarmStatus(): ResponseEntity<String>
+registerToken(body): ResponseEntity<String>
+triggerAlarm(): ResponseEntity<String>
DeviceManagerServiceImpl
State (cache & deps)
-rooms: Map<String, Set<String>>
-groups: Map<String, Set<String>>
-devices: Map<String, Device>
-channels: Map<String, ManagedChannel>
-repo: DeviceEventRepository
-rgRepo: RoomGroupRepository
-aliasRepo: DeviceAliasRepository
-restTemplate: RestTemplate
-alarmActive, alarmTriggered: boolean
Operazioni principali (estratto)
+registerDeviceHttp(id, type, addr, port): RegisterDeviceResponse
+turnOnDevice(id): String
+getStatusDevice(id): String
+setProgramDevice(id, program): String
+createRoom(roomId): String
+addDeviceToRoom(roomId, deviceId): String
+renameRoom(oldId, newId): String
+createGroup(groupId): String
+renameGroup(oldId, newId): String
+renameDevice(oldId, newId): String
+setDeviceDisplayName(id, name): String
+activateAlarm(enable): String
#scheduledSensorCheck(): void
logs
rooms/groups
aliases
get DS
gRPC calls
logging
DeviceEventRepository
+insertEvent(...)
+listRecent(limit): List<DeviceEvent>
RoomGroupRepository
+createRoom(roomId)
+addDeviceToRoom(roomId, deviceId)
+renameRoom(oldId, newId)
+createGroup(groupId)
+addDeviceToGroup(groupId, deviceId)
+renameGroup(oldId, newId)
+renameDeviceEverywhere(oldId, newId)
DeviceAliasRepository
+loadAll(): Map<String,String>
+upsertAlias(deviceId, displayName)
PgDataSource
+get(): DataSource
gRPC stubs (com
smarthome
proto)
Device
+deviceId: String
+deviceType: String
«gRPC» LightServiceGrpc
«gRPC» OvenServiceGrpc
«gRPC» DishwasherServiceGrpc
«gRPC» MotionSensorServiceGrpc
«gRPC» AirConditionerServiceGrpc
«gRPC» ThermostatServiceGrpc
«gRPC» SolarPanelServiceGrpc
«gRPC» BlindServiceGrpc
RegisterDeviceResponse

The diagram illustrates the layered architecture of the device management system.

- **API Layer**: contains the controllers that expose REST services to the outside (DeviceApiController and AlarmApiController). These are the main entry points for clients or third-party applications.
- **Service Layer**: at the core of the application logic is DeviceManagerServiceImpl. This class coordinates requests from the controllers, applies business rules, and delegates operations to the underlying modules.
- **Persistence**: includes repositories (DeviceEventRepository, RoomGroupRepository, DeviceAliasRepository) and the data source (PgDataSource), responsible for storing events, rooms, aliases, and managing connections to the relational database.
- **Integrations**: manages interactions with external systems, in particular gRPC microservices for various devices and the Expo Push API for notifications.

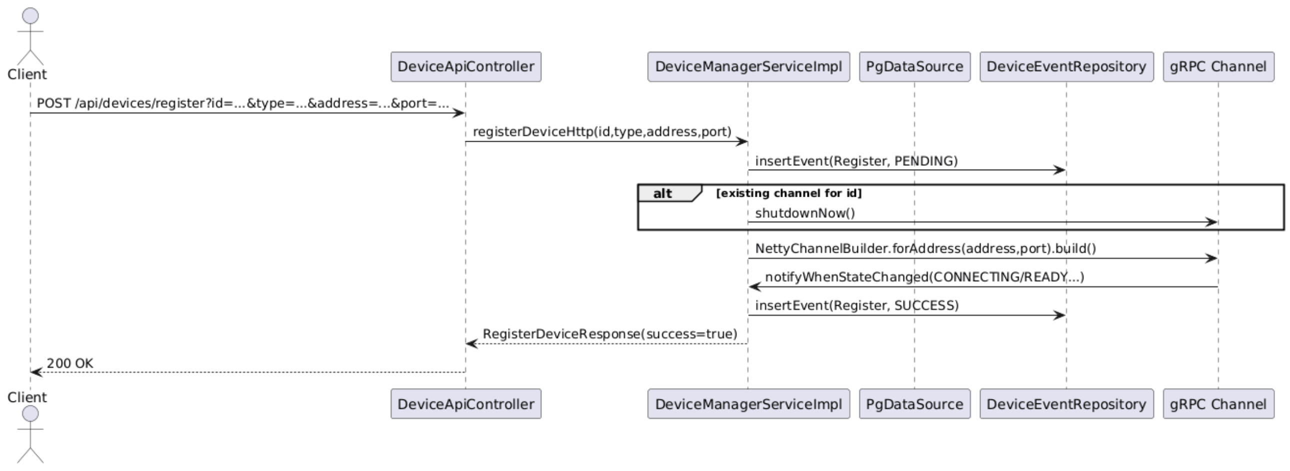


The internal class diagram provides a **detailed UML class view** of the main components.

- The **api** package defines the controllers, with public methods for device registration, status retrieval, creation and renaming of rooms and groups, alias management, and event logging.

- The DeviceManagerServiceImpl class plays a central role: it maintains internal data structures (maps of rooms, groups, devices, and gRPC channels), encapsulates repository references, and coordinates calls to external microservices. Its methods cover all core functions, from registering new devices to enabling alarms.
- The **persistence** classes define basic CRUD operations: DeviceEventRepository for logging, RoomGroupRepository for managing spaces, and DeviceAliasRepository for alias mapping.
- The **gRPC stubs** section includes classes generated from proto files (LightServiceGrpc, DishwasherServiceGrpc, MotionSensorServiceGrpc, ThermostatServiceGrpc, etc.), representing integration points with vertical microservices for each device type.
- Additional entities include Device (encapsulating ID and type) and RegisterDeviceResponse (the result of registering a new device).

Interaction with gRPC connectivity

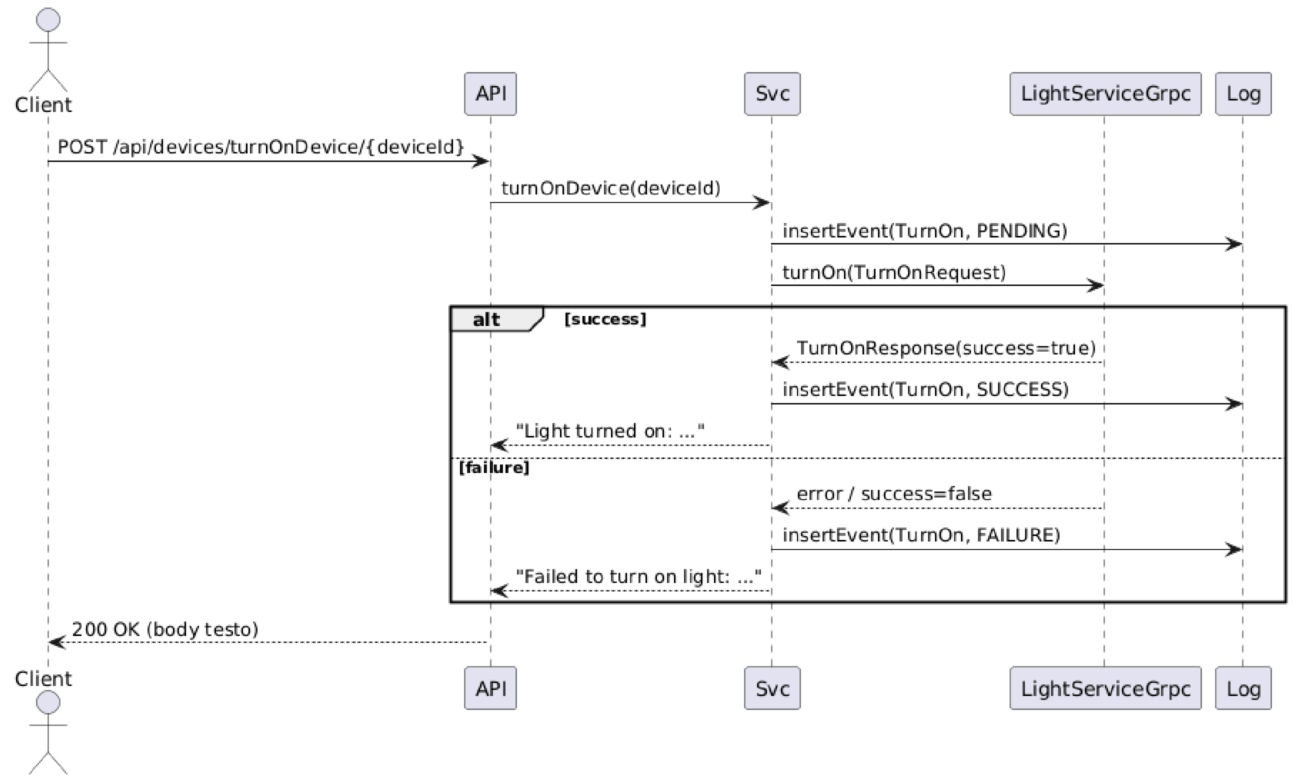

Client
DeviceApiController
DeviceManagerServiceImpl
PgDataSource
DeviceEventRepository
gRPC Channel
POST /api/devices/register?id=...&type=...&address=...&port=...
registerDeviceHttp(id,type,address,port)
insertEvent(Register, PENDING)
alt
[existing channel for id]
shutdownNow()
NettyChannelBuilder.forAddress(address,port).build()
notifyWhenStateChanged(CONNECTING/READY...)
insertEvent(Register, SUCCESS)
RegisterDeviceResponse(success=true)
200 OK

The first sequence diagram illustrates the process of registering a new device via HTTP and establishing its gRPC connection.

- The **Client** sends a POST /api/devices/register request with device parameters (id, type, address, port).
- The **DeviceApiController** forwards the request to the service (DeviceManagerServiceImpl), which begins the registration by inserting a **PENDING** event into the DeviceEventRepository.
- If a gRPC channel for the given device ID already exists, it is shut down before creating a new one.
- A new channel is built using NettyChannelBuilder, and connection state changes are tracked (CONNECTING / READY).

- Once the connection is established, a **SUCCESS** event is logged, and the service returns a RegisterDeviceResponse(success=true) to the controller.
- Finally, the controller responds to the client with 200 OK.

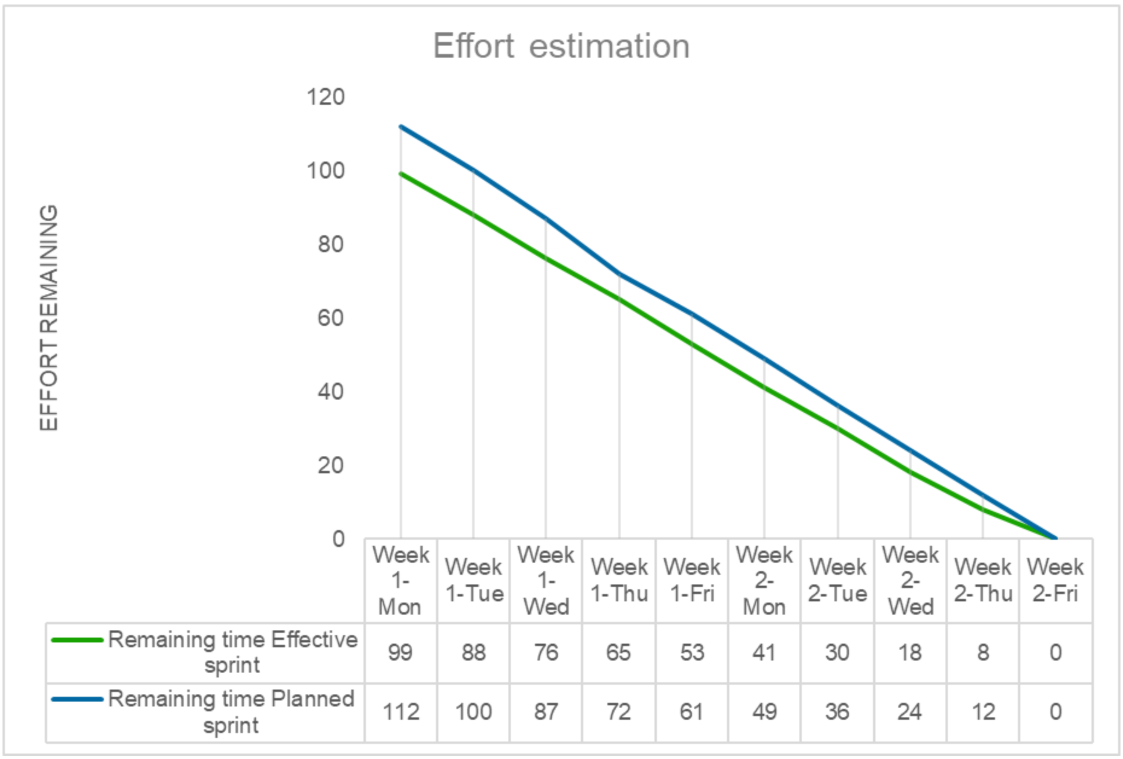
This flow highlights how the system transitions a device from an HTTP request into an active gRPC connection, while keeping track of registration events in the persistence layer.


Client
API
Svc
LightServiceGrpc
Log
POST /api/devices/turnOnDevice/{deviceId}
turnOnDevice(deviceId)
insertEvent(TurnOn, PENDING)
turnOn(TurnOnRequest)
alt
[success]
TurnOnResponse(success=true)
insertEvent(TurnOn, SUCCESS)
"Light turned on: ..."
[failure]
error / success=false
insertEvent(TurnOn, FAILURE)
"Failed to turn on light: ..."
200 OK (body testo)

The second sequence diagram shows the sequence for turning on a light device.

- The **Client** sends a POST /api/devices/turnOnDevice/{deviceId} request.
- The **API controller** invokes the service (turnOnDevice(deviceId)), which logs a **PENDING** event and forwards a gRPC TurnOnRequest to the LightServiceGrpc.
- Two alternative outcomes are modeled:
    - **Success**: the service receives a TurnOnResponse(success=true), logs a **SUCCESS** event, and returns a confirmation message (“Light turned on”).
    - **Failure**: the service receives an error or a success=false response, logs a **FAILURE** event, and returns an error message (“Failed to turn on light”).
- In both cases, the API responds to the client with 200 OK, including the message in the response body.

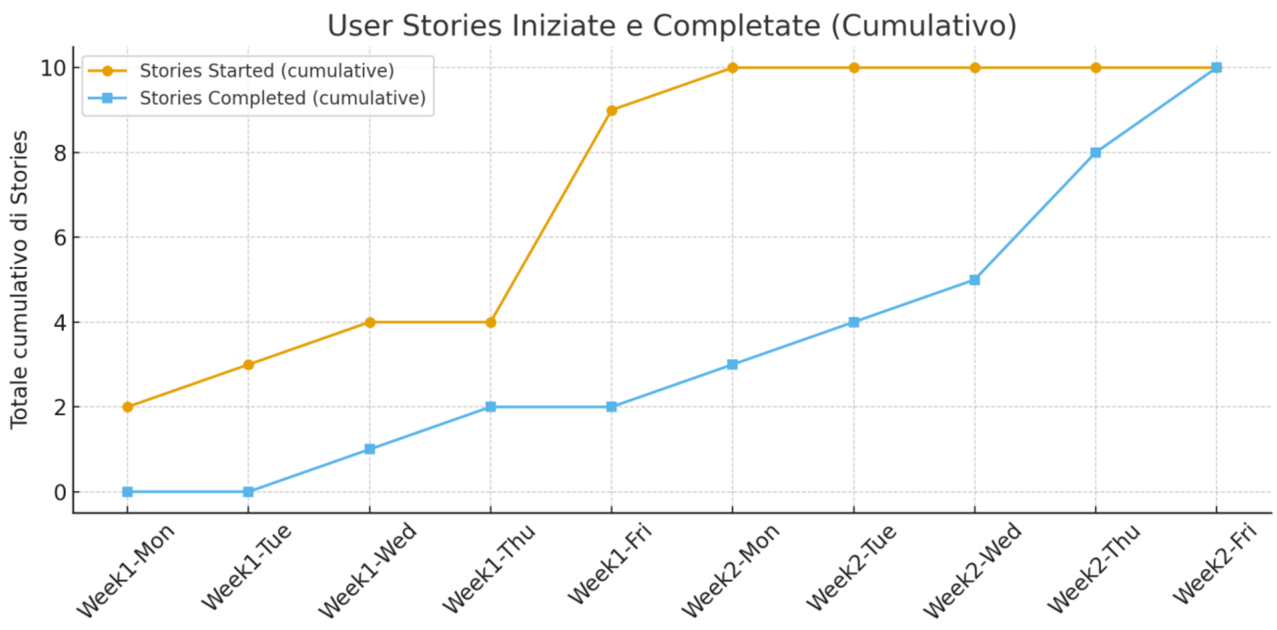
SCRUM and SPRINT analisys



	Week 1-Mon	Week 1-Tue	Week 1-Wed	Week 1-Thu	Week 1-Fri	Week 2-Mon	Week 2-Tue	Week 2-Wed	Week 2-Thu	Week 2-Fri
Remaining time Effective sprint	99	88	76	65	53	41	30	18	8	0
Remaining time Planned sprint	112	100	87	72	61	49	36	24	12	0

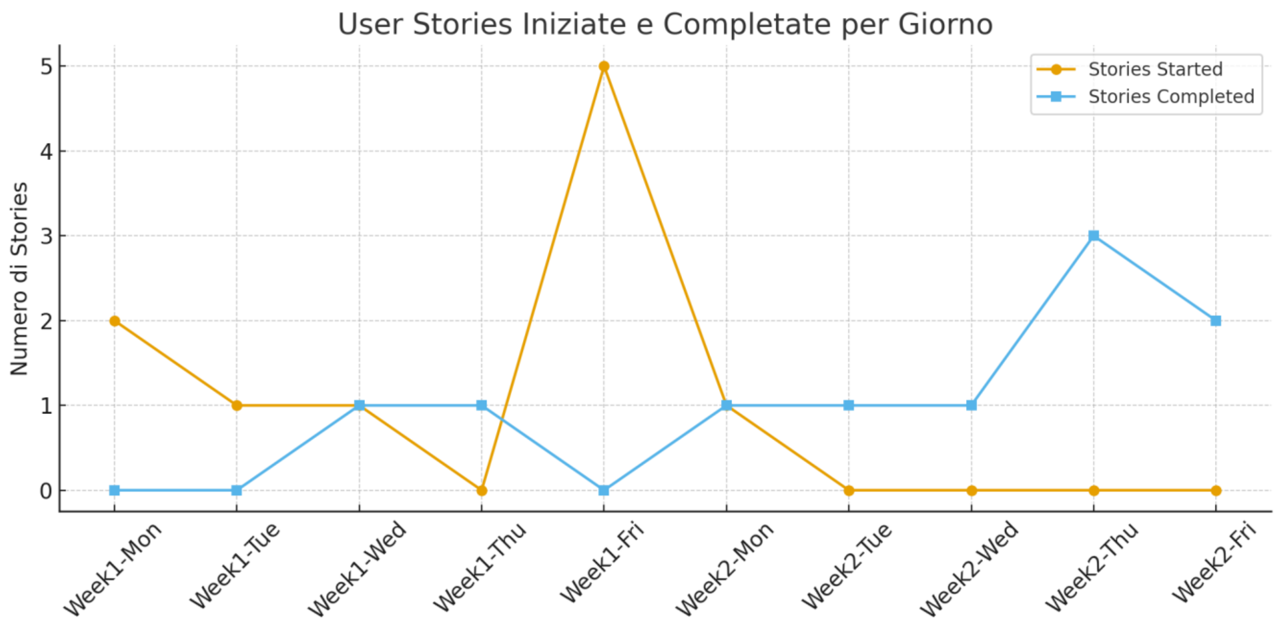
The effort chart shows the **burndown of remaining effort** over the two weeks of the sprint.

- The **blue line** represents the planned effort reduction (ideal sprint progression).
- The **green line** shows the actual effort consumed day by day.
- The chart highlights that the team's effective progress was consistently ahead of the original plan, with the sprint closing slightly earlier than expected.



The cumulative stories chart tracks **user stories cumulatively** throughout the sprint.

- The **orange line** represents the total number of stories started.
- The **blue line** shows the total number of stories completed.
- The gap between the two lines represents work in progress (stories started but not yet finished).
- By the end of the sprint, the lines converge, showing that all ten planned stories were completed.



The last chart provides a **daily view** of stories started and completed.

- Peaks in the orange line (stories started) indicate when new work was taken on, with a spike on Week 1–Friday.
- The blue line (stories completed) shows when value was delivered, with multiple completions clustered towards Week 2–Thursday and Friday.
- The distribution highlights that while stories were started unevenly, the team managed to close them efficiently before the sprint deadline.