ChatGPT

Building a Modern JSON Calendar Format: Lessons from iCalendar

Current State of Calendar Data Formats

iCalendar (RFC 5545): The dominant standard for calendar data exchange is the iCalendar format (often with `.ics` files). It's a plain-text format defined in RFC 5545 that can represent events, to-dos, journal entries, and free/busy information [1]. In iCalendar, a calendar is represented as a collection of structured lines of text (each called a *content line*). For example, an event (VEVENT) is described by lines like `SUMMARY:Meeting with Bob` or `DTSTART:20250620T100000Z`. This text-based structure, while human-readable in a pinch, was designed for robustness and interoperability rather than ease of use by developers or AI. Over time, various extensions (additional RFCs) have updated iCalendar to support new features and use cases. The core format, however, remains anchored in a 1990s-era design – centered on calendaring objects in text form and meant to be independent of any particular system [1].

iCalendar Extensions and Updates: Since RFC 5545's publication, several RFCs have extended the format:

- **Non-Gregorian Recurrence (RFC 7529):** Introduced support for non-Gregorian calendars in recurrence rules [2]. Originally, iCalendar assumed the Gregorian calendar exclusively, with an unused `CALSCALE` property intended for other calendars. RFC 7529 updates the recurrence rule (RRULE) syntax to include an `RSCALE` parameter and related features so events can repeat according to calendars like Hebrew or Chinese lunar calendars [3] [4] (while still keeping date values in Gregorian for interoperability).

- **New Properties for Events (RFC 7986):** Added a set of new properties to enrich calendar data [5]. Notable examples include a `CONFERENCE` property for specifying conference call details (e.g., video meeting links), an `IMAGE` property for associating an image with an event, a `COLOR` property for color-coding calendars, and a `NAME` property to name a calendar (useful when sharing calendars) [6]. These extensions make iCalendar more suitable for modern applications (for instance, integrating with conferencing systems).

- **Event Publishing and Structured Data (RFC 9073):** Focused on event publishers (e.g. public event feeds or social networks) by introducing new components and properties [7]. It defined structured location (`VLOCATION`), participant (`VPARTICIPANT`), and resource (`VRESOURCE`) components, allowing events to reference detailed sub-objects instead of just free-form text. It also added a `STRUCTURED-DATA` property to embed JSON or XML data within an event [7] – essentially a way to include machine-readable payloads relevant to the event (for example, a JSON snippet with extra metadata). These features aim to make event data more interoperable and richly descriptive.

- **Alarm Extensions (RFC 9074):** Improved the handling of reminders/alarms (VALARM) [8]. Classic iCalendar alarms could trigger at a set time (e.g., 10 minutes before an event) and perform actions like DISPLAY (pop-up) or EMAIL. RFC 9074 adds support for *acknowledging* alarms (recording that an alarm was dismissed by the user) and *snoozing* (postponing an alarm) in a standardized way [9]. It introduces properties like `ACKNOWLEDGED` (with a timestamp of dismissal) and a mechanism to specify snooze intervals, which previously had no official representation (clients resorted to non-standard hacks) [10]. It also allows alarms to have unique IDs and even defines a "proximity" trigger – an alarm that can fire when nearing a location, not just based on time (reflecting smartphone capabilities) [11] [12].

- **Calendar Availability (RFC 7953):** Defined a new component VAVAILABILITY to publish typical availability patterns [13]. This allows users to formally specify times they are available or unavailable (e.g., office hours, vacation times) on their calendar, which can be used when someone else checks your free/busy status. It's an adjunct to the free/busy system: instead of calculating free times solely from events, one can also account for "available" blocks defined by VAVAILABILITY [13].

- **Calendar Relationships (RFC 9253):** Enhanced the ability to link related calendar entries [14]. It updates the `RELATED-TO` property (which links one event to another, like a follow-up meeting or a series of tasks) by adding relation types (e.g., "PARENT"/"CHILD", "NEXT"/"PREV"). It also adds new properties: `LINK` (to directly hyperlink to external resources or related items), `CONCEPT` (to group items under an arbitrary concept or theme), and `REFID` (a reference identifier to correlate components across systems) [14]. These let calendar data be more interconnected – useful in complex workflows or linking to contact info, documents, or tasks.

- **Protocol and Data Exchange:** Alongside the data format, key standards exist for **scheduling** and **sync**. The iCalendar **Transport-Independent Interoperability Protocol (iTIP, RFC 5546)** defines how calendar data is used to schedule meetings by email or other means [15]. It describes scheduling methods (like REQUEST, REPLY, CANCEL) that allow organizers and attendees to coordinate (e.g., sending invitations and responses) [16]. For syncing data with servers, **CalDAV (RFC 4791)** is widely used – a WebDAV-based protocol where the server stores iCalendar objects and clients use HTTP methods to fetch/update them. CalDAV scheduling (RFC 6638) builds on iTIP to let servers handle meeting invitations. There's also iCalendar **Email-based Interop (iMIP, RFC 6047)** for sending iTIP messages via email. These protocols ensure that calendar entries created in one system can be transferred or synced to another. However, they inherit the complexity of iCalendar's data model.

Alternative Formats (XML/JSON): Recognizing the awkwardness of the ics text format, the community created alternate representations: - *xCal (RFC 6321)* – an XML representation of iCalendar. - *jCal (RFC 7265)* – a JSON representation of iCalendar data [17] [18]. jCal essentially converts each iCalendar component and property into JSON arrays and objects, maintaining the same structure and naming (for example, an event becomes a JSON object with a property array like `["uid", {}, "text", "XYZ-123"]` for UID). This makes it easier for web applications to consume calendar data without writing a full .ics parser, but it's basically a direct mapping from the text format, so it carries over all of iCalendar's conceptual baggage (including legacy properties, the need to parse RRULE strings, etc.). - *JSCalendar (RFC 8984)* – a more modern JSON calendar format designed by the IETF after jCal. Rather than a one-to-one mapping, JSCalendar defines a new JSON schema for calendar events, stripping out some obsolete concepts and using more intuitive keys. For example, it uses `"title"` instead of `"SUMMARY"`, and represents recurrences and participants as structured sub-objects. It was made to be used with the JMAP protocol for syncing (a contemporary alternative to CalDAV). JSCalendar is a close analogue to what you're envisioning: it's **JSON-**

native and designed for both machine processing and human readability. It coexists with iCalendar – intended for new systems or as a translation layer – since iCalendar is still widely entrenched.

In summary, **the current state** is that iCalendar (RFC 5545) remains the common format for calendar data interchange across different systems (Google Calendar, Outlook, Apple Calendar, etc.), thanks to decades of adoption. It's a text format with a lot of capabilities, extended by multiple RFCs to address shortcomings (from recurrence in other calendars to video-conference info in events). However, it also carries legacy features that aren't commonly used in modern personal calendaring. New efforts like JSCalendar show an appetite for a *fresh, JSON-based approach*, but those are still emerging. This context is the starting point for thinking about a new "JSON Calendar" format that aims to be simpler and more in line with today's technology (and AI applications).

Legacy Features to Remove (Simplifying the Format)

To build a modern, minimal calendar format, we should identify which parts of iCalendar are **"legacy" or extraneous** for personal calendar use. These can be left out to streamline the JSON calendar:

- **To-Dos (VTODO) and Journals (VJOURNAL):** iCalendar was designed to support not just events but also task management and journal/daily diary entries [19]. In practice, tasks (to-do lists) and journal entries are handled by specialized apps or not used at all by typical users. Including them makes the format more complex (each has its own properties and scheduling semantics). For a personal calendar focusing only on events (meetings, reminders, appointments), we can drop VTODO and VJOURNAL entirely. This means we don't need to worry about task-specific properties like percent-complete or due dates, or journal metadata – a significant simplification.

- **Free/Busy Components (VFREEBUSY):** VFREEBUSY components let a user or server send a block of time slots that are busy or free (often used in scheduling to negotiate meeting times). Today, free/busy is usually determined on the fly by querying a calendar server (e.g., via CalDAV or an API) rather than exchanging static files. And for personal use, you typically know your own free/busy times by looking at your events. So, a minimal format can omit VFREEBUSY as a separate object. If needed, free/busy info can be derived from event data or an availability schema (like VAVAILABILITY) rather than a distinct component.

- **VTIMEZONE (Embedded Time Zone Definitions):** In .ics files, each time zone used by events can be described by an embedded VTIMEZONE component containing the timezone's rules (daylight savings transitions, etc.). This was crucial when clients might not have up-to-date timezone databases. However, it greatly bloats the data (a single timezone definition can be dozens of lines of recurrence rules for DST). A modern approach could assume clients share a standard time zone database (like the IANA TZ database). Thus, instead of embedding full definitions, the JSON format can just store the timezone identifier (e.g., `"America/New_York"`). Removing VTIMEZONE components cuts a lot of noise. If a client/AI needs to calculate times, it likely has access to a library for that or could call a service. (In cases where offline use is needed, one could cache timezone info separately, rather than in every calendar file.)

- **CALSCALE (Calendar Scale):** This property in iCalendar was meant to allow non-Gregorian date systems by switching the entire calendar's scale (e.g., to Hebrew or Chinese calendar) [20]. In reality, it's almost always "GREGORIAN" and never changed, because a global switch proved impractical (mixing events from different cultures in one file was impossible under that model) [21]. RFC 7529's solution was to keep CALSCALE as Gregorian and extend RRULEs for other calendars. For a new format, we can simplify by assuming Gregorian dates for the data (at least initially). Non-Gregorian recurrences, if needed, can be handled via an extension similar to RSCALE. But we certainly don't need a CALSCALE property at the top-level; the default can be implicitly Gregorian, avoiding an unused legacy setting.

- **Deprecated or Uncommon Properties:** iCalendar has a few things that have been deprecated or just rarely used:

- **EXRULE** (Excluded Recurrence Rule) was a way to specify an exclusion recurrence pattern; it was deprecated in RFC 5545 in favor of using EXDATE. A new format wouldn't reintroduce EXRULE.
- **REQUEST-STATUS** properties on replies (to convey errors or confirmations in scheduling) – not typically used outside of enterprise scheduling flows.
- **CLASS (PUBLIC/PRIVATE/CONFIDENTIAL)** classification of events – many clients ignore this, and privacy is often handled at the sharing permission level rather than per event. This could be left out or simplified (maybe a simple boolean for private/public if needed).
- **TRANSP (Transparent/Opaque)** – indicates if an event blocks free time. This might be relevant to keep (e.g., you may want some events to be "FYI" and not mark you busy), but it could be represented in a simpler way (a flag in the event).

- **Status values** like TENTATIVE – not all systems use Tentative versus Confirmed. One could opt to drop Tentative or handle it via participant status (e.g., if *you* haven't decided on attending an event, maybe it's marked differently). Tentative events might be represented just as normal events with perhaps a tag.

- **Sequence Numbers and Scheduling Fields:** In iCalendar, whenever you update an event that has been sent out to participants, you increment its `SEQUENCE` number to indicate a new version. Attendees use this to know if they're looking at the latest update [22]. There's also a `DTSTAMP` indicating when the event was created/sent. In a modern environment, especially for a personal calendar, we might not need these at all – or we rely on a simpler mechanism (like a last-modified timestamp or an ETag if on a server). If our JSON format is used in a client-server model, the server's API can handle versioning; if it's used peer-to-peer, perhaps a single modified timestamp or even the JSON document's own version control (in git or similar) could serve the purpose. Thus, we could drop `SEQUENCE` and `DTSTAMP` to reduce noise, unless we plan to incorporate full iTIP-style scheduling.

- **Attendee Delegation and Complex Scheduling Options:** iCalendar/iTIP allow an attendee to delegate an event to someone else, or for organizers to specify complex recurrence overrides in meeting proposals. These features are infrequently used. A minimal format might ignore delegation entirely (assuming the user will directly invite who needs to attend). Also, iTIP's rich set of methods (COUNTER to propose a new time, REFRESH to request latest info, etc.) could be pared down. If our focus is not on designing the transport protocol, we simply need to ensure the event data can capture attendees and their replies – not necessarily the whole workflow states. This ties more to protocol than format, but it means we don't necessarily need fields for "counter proposal" or "delegated from" in our data model from the get-go.

In essence, by **removing these legacy or edge-case features**, the calendar format becomes leaner. We stick to core concepts of events (with time, location, description, etc.), basic recurrence, and perhaps invitees. We let modern systems handle things like free/busy calculation or synchronization logic outside the data format. The result is a simpler schema that's easier to understand and maintain, and less prone to misuse or confusion. This lean approach particularly benefits AI or any new implementation because there are fewer moving parts to parse and reason about.

(We will still want to ensure that the essentials needed for personal calendaring – like recurring events, notifications, and sharing events with others – are retained, but implemented in a cleaner way.)

Event Representation in iCalendar vs JSON

iCalendar Event Structure: In iCalendar, an event is represented by a **VEVENT** component. It's essentially a block of text lines between `BEGIN:VEVENT` and `END:VEVENT` within the calendar file. Each line is a property of the event, often with parameters. Key properties include:

- `UID`: A globally unique identifier for the event (usually a random or GUID-like string) [23]. This is crucial for syncing and updating events across systems.
- `DTSTART`: The start date/time of the event.
- `DTEND` or `DURATION`: The end date/time, or a duration from the start. One of these is required for a timed event (for all-day events, DTSTART (date) and DTEND (date) are used without time). If DTEND is omitted, some systems infer the event lasts a default duration (or zero-length for an instant).
- `SUMMARY`: A one-line summary or title of the event (what most users see as the event name).
- `DESCRIPTION`: A longer description or notes (optional).
- `LOCATION`: Where the event takes place (could be a physical address or place name, or even something like "Online").
- `ORGANIZER`: Who organized the event (often an email address in a `mailto:` URI form). For personal calendar events this might often be the owner of the calendar.
- `ATTENDEE`: Used to list participants. Each ATTENDEE is a separate property line with a mailto: URI and parameters for the attendee's name, role (CHAIR, REQ-PARTICIPANT, etc.), status (ACCEPTED, TENTATIVE, etc.), and so on. For example: `ATTENDEE;CN=Alice;PARTSTAT=ACCEPTED:mailto:alice@example.com` describes an attendee Alice who accepted.
- `STATUS`: The status of the event itself – e.g., CONFIRMED, TENTATIVE, or CANCELLED. This is set by the organizer typically (Cancelled means the event is called off; Tentative might mean "penciled in").
- `CLASS`: Classification (PUBLIC or PRIVATE), indicating sensitivity of the event.
- `TRANSP`: Transparency – OPAQUE means it blocks your free/busy as busy, TRANSPARENT means it doesn't (e.g., an FYI event that shouldn't make you appear unavailable).
- `CATEGORIES`: Tags or keywords for the event (e.g., "Vacation" or "Finance/Review"). This is a list of labels for grouping or coloring events in UI.
- `VALARM`: Alarm subcomponents (explained below).
- Other optional fields: e.g., `URL` (a URL associated with the event, maybe a reference or meeting link), `LAST-MODIFIED` (timestamp of last update), `CREATED` (when it was originally created), `SEQUENCE` (revision number, as noted), `RECURRENCE-ID` (if this VEVENT is an exception to a recurrence series, this indicates which instance it corresponds to).

The **content of these properties** is mostly text. Some have specific formats (dates in YYYYMMDD or date-times in YYYYMMDDThhmmssZ format for UTC). Properties can have *parameters* – e.g., `ATTENDEE` has parameters for role, status, etc., and `DTSTART` might have a `TZID=America/Chicago` parameter to indicate it's in a specific timezone.

One challenge: iCalendar property names are terse (often all-caps abbreviations) and the format requires careful parsing (splitting by colon and semicolon, respecting line folding rules, etc.). For example, an event might contain:

```
DTSTART;TZID=Europe/Stockholm:20250620T124500
DTEND;TZID=Europe/Stockholm:20250620T134500
SUMMARY:Lunch with CEO
LOCATION:Office Restaurant (Floor 5)
ATTENDEE;CN=Jane Doe;PARTSTAT=ACCEPTED:mailto:jane@example.com
ATTENDEE;CN=John Doe;PARTSTAT=TENTATIVE:mailto:john@example.com
```

This is reasonably compact, but not immediately intuitive without knowledge of the spec.

Alarms (VALARM): Within a VEVENT (or VTODO), you can include a reminder by adding a sub-component VALARM. The VALARM has its own properties: - `TRIGGER`: when the alarm goes off (could be relative like "-PT10M" for 10 minutes before the event, or an absolute time). - `ACTION`: what happens (DISPLAY for on-screen alert, AUDIO for a sound, EMAIL to send an email, etc.). - For DISPLAY, a `DESCRIPTION` gives the text of the alert. For EMAIL, you'd have who to email and the subject/body. - Optionally, `REPEAT` and `DURATION` can specify if an alarm repeats (snooze feature, sort of).

However, classical iCalendar had no standard way to say "the user dismissed this alarm at 12:01". This is what RFC 9074 addresses by adding: - an `ACKNOWLEDGED` property (Date-Time when the alarm was acknowledged/dismissed) [24] [11], - allowing VALARMs to have a UID (so that, for example, a client can refer to "Alarm 123" when syncing alarm state) [25], - clarifying how to handle multiple alarms and their relationships (the "snoozing" concept is essentially repeating alarms or new alarms that relate to the original – RFC 9074 provides guidance and a property to link a snooze alarm to its parent) [12] [26], - adding a "PROXIMITY" trigger type to allow location-based triggers (e.g., alert when I arrive at X). This is beyond the scope of basic personal calendaring for many users, but it's a forward-looking feature considering mobile usage.

Enhanced Event Metadata: Modern extensions have tried to keep iCalendar relevant: - RFC 7986's `CONFERENCE` property allows specifying one or more conference call details [27]. It usually includes a URI (like a Zoom or Teams link) and possibly a `FEATURE` parameter (to note if it's video, phone, etc.) [28] [29]. For example: `CONFERENCE;VALUE=URI;FEATURE=VIDEO:https://zoom.example.com/abc123`. - The `IMAGE` property can be used to attach an image (by URI or embedded data). A conference or event can also have an image (like a speaker's photo or an event flyer) [28]. There's also a `DISPLAY` parameter for images (like thumbnail, banner, etc.). - RFC 9073's structured locations/participants: Instead of just a plain text LOCATION, a VLOCATION component can carry structured address info or coordinates, and the VEVENT can reference it (via a property link). Similarly, VPARTICIPANT can hold info about a person/resource (email, name, perhaps even avatar or role details) and the event's ATTENDEE property can reference that. This level of indirection is not widely implemented yet, but it points to a desire to treat locations and people as first-class linked data, rather than repeating text in each event [30] [31]. - `STRUCTURED-DATA`: Allows embedding data such as a JSON payload in a VEVENT [7]. For instance, an event for a concert might include structured-data with a JSON containing setlist or performer details, which a smart client could use to display more info. It's basically a way to attach domain-specific data without the calendar format needing to understand it – useful for publishers of events.

How a JSON Format Would Represent Events: A JSON-based calendar format would naturally represent an event as a JSON object (dictionary) with keys corresponding to the above properties, but with more developer-friendly naming and typing. For example, an event might look like:

```
{
  "id": "123456-abcde",
  "title": "Lunch with CEO",
  "description": "Discuss quarterly results.",
  "start": { "dateTime": "2025-06-20T12:45:00", "timeZone": "Europe/
Stockholm" },
  "end":   { "dateTime": "2025-06-20T13:45:00", "timeZone": "Europe/
Stockholm" },
  "location": {
      "name": "Office Restaurant (Floor 5)"
  },
  "attendees": [
      { "name": "Jane Doe", "email": "jane@example.com", "status": "accepted" },
      { "name": "John Doe", "email": "john@example.com", "status": "tentative" }
  ],
  "organizer": { "name": "Me", "email": "me@mycalendar.com" },
  "recurrence": { ... },
  "alarms": [ ... ]
}
```

This is an illustrative snippet (borrowing concepts from JSCalendar and common JSON calendar APIs). The benefits are clear:

- **Clarity:** Keys like `"title"` and `"start"` are self-explanatory compared to `SUMMARY` and `DTSTART`. An AI or developer reading this JSON can guess the meaning without a spec.
- **Structured Dates:** Instead of a single string "20250620T124500" that needs parsing, the JSON could use ISO 8601 or split the date/time and timezone for clarity. (In the example, I showed a structure for dateTime + timeZone, but it could also be one combined string with timezone info.)
- **Typed Values:** JSON knows about numbers, booleans, etc. In iCalendar, everything is text that needs interpretation. For instance, in JSON we could use a boolean `{"allDay": true}` for all-day events instead of relying on the convention of date format vs date-time format in DTSTART. Or an integer for sequence if we had one. This reduces ambiguity.
- **Attendees as Array:** This is much cleaner than multiple ATTENDEE lines. Each attendee can have properties in a single object. The status is just a field rather than hidden in a parameter. We could even nest things like a response comment if needed.

- **Location as Object:** We can extend location with an address or coordinates easily if needed (e.g., `"coords": [59.33, 18.06]` for latitude/longitude). In iCalendar, one would have to use either a geo property (separate) or embed info in the LOCATION text or use the STRUCTURED-DATA approach.
- **Alarms as Array of Objects:** Each alarm can be one object with when to trigger and how. E.g., `{ "offset": "-PT10M", "action": "display", "message": "Reminder: Lunch with CEO in 10 minutes" }`. This is straightforward to manipulate – an AI agent could decide to reschedule the alarm by just changing the offset field.

Interoperability of Fields: The JSON format can also incorporate or drop certain fields: - If we drop `STATUS` (CONFIRMED/TENTATIVE) at the event level, we might indicate a tentative hold by some other means (like a tag or just not confirming attendance for your own event). - `CLASS` (privacy) might not be needed if we assume all events in a personal calendar default to one behavior, or it could be a simple `{"privacy": "public"}` or a boolean. - One thing to consider is mapping to external systems: if we ever needed to convert back to iCalendar, we should not lose essential info. But since we're not focusing on backwards compatibility, we have flexibility.

RFC References in JSON Context: The iCalendar enhancements give hints for what a JSON format might include. For example, the new properties from RFC 7986 (like `CONFERENCE`, `IMAGE`) would just become additional fields in JSON (perhaps an `onlineMeetingUrl` or an array of `conferenceDetails`). The idea of structured participants from RFC 9073 is naturally achieved in JSON with nested objects as shown. The linking concepts from RFC 9253 (LINK, RELATED-TO improvements) could also be represented as, say, an array of related event IDs or URLs in the JSON (e.g., `"related": [ {"rel": "parent", "href": "..."} ]` or simpler if just internal references).

In short, **events in a JSON calendar format** would carry essentially the same information as an iCalendar VEVENT, but in a cleaner hierarchy. This benefits both humans (e.g., easier to read a JSON dump of an event) and machines (no need to write a custom parser or deal with decoding text encodings). An AI system could ingest a JSON event object and have direct access to all fields as data, rather than needing to know iCalendar's peculiar syntax rules. The rich features added by various RFCs (conference info, structured locations, etc.) can be integrated in an intuitive way (just additional keys/objects) without the rigidity of iCalendar's line-based format.

Recurrence Rules and Exceptions

Recurrence in iCalendar: The iCalendar spec allows events to repeat according to rules. This is primarily done through the `RRULE` property on an event, and sometimes supplemented by `RDATE` (extra dates) and `EXDATE` (exception dates). The RRULE syntax is quite powerful – it can express complex patterns but is essentially a mini-language encoded in a single line. For example:

- `RRULE:FREQ=WEEKLY;INTERVAL=2;BYDAY=MO,WE,FR` means the event repeats every 2 weeks on Monday, Wednesday, and Friday [18].
- `RRULE:FREQ=YEARLY;BYMONTH=12;BYDAY=MO;BYSETPOS=2` would mean "the second Monday of December every year" (BYSETPOS=2 picks the second occurrence of BYDAY in that month).
- You can have COUNT or UNTIL to stop the recurrence after a certain number or date.

While compact, these rules are not immediately obvious to casual readers (or to software that isn't specifically programmed to parse them). Any AI or program has to parse this string and interpret the recurrence semantics.

Exceptions: If an event repeats, there are often changes or cancellations for specific instances. iCalendar handles this by: - **EXDATE:** listing specific dates/times that should be skipped (not occur). E.g., `EXDATE;TZID=Europe/Stockholm:20251225T124500` would exclude an occurrence on Christmas (maybe you're not having a weekly meeting that day). - **Recurrence ID (RECURRENCE-ID):** to modify a particular occurrence, iCalendar creates a separate VEVENT with the same UID and a RECURRENCE-ID property equal to the date/time of the occurrence being overridden. For example, if normally an event happens every Monday, but on one particular date you want to change the time or details, you include a new VEVENT with RECURRENCE-ID equal to that date. That "exception event" can have a different start/end or even STATUS:CANCELLED if the event is cancelled that day. When rendering the calendar, clients merge these exceptions with the main recurring event rule.

This mechanism works but is quite heavy. It means a recurring series can turn into multiple VEVENTs in an .ics file (one for the series pattern, plus one for each changed instance). Reconstructing the full set of event instances requires combining them: taking the RRULE from the main event, removing any EXDATEs, and substituting any instance that has a RECURRENCE-ID override.

Non-Gregorian Recurrences (RFC 7529): As mentioned earlier, iCalendar originally only did Gregorian recurrences. RFC 7529 extends the RRULE syntax with new elements to support other calendars [32]. It introduced: - `RSCALE` parameter (e.g., RSCALE=HEBREW or CHINESE) to specify the calendar system for the rule [4]. - `SKIP` parameter to specify how to handle dates that don't exist in certain years (like the 30th of a month in a calendar that might have shorter months in some years, or leap month scenarios) [33] [32]. Values for SKIP could be FORWARD or BACKWARD (choose next valid date or previous) or OMISSION (skip occurrence). - This allows, for example, an event every year on 10th day of the Chinese New Year (lunar calendar) to be represented in an iCalendar file that is otherwise Gregorian.

For many users who stick to Gregorian, this might not be used, but it's good to know how recurrence expanded to cover global needs. If designing a JSON format, one could incorporate something similar if needed (perhaps a field like `"calendar": "hebrew"` inside the recurrence object), but one might leave it out initially if targeting a minimal viable set (and simply note that dates are Gregorian by default).

Simpler Recurrence in JSON: One major opportunity with JSON is to represent recurrence rules in a structured way instead of a single string. We could break out the components of the rule: - `"frequency": "weekly"` instead of `FREQ=WEEKLY`. - `"interval": 2` for INTERVAL=2. - `"daysOfWeek": ["MO","WE","FR"]` for BYDAY=MO,WE,FR. - If something like second Monday of December: `"frequency": "yearly"`, `"months": [12]`, `"weekday": "MO"`, `"weekdayOccurrence": 2` (meaning 2nd occurrence of Monday). - An end condition could be `"count": 10` or `"until": "2025-12-31T23:59:59Z"`.

This approach is more verbose than the single RRULE line, but it's self-explanatory and eliminates the need for a custom parser. An AI could easily reason about such a JSON structure (e.g., it could answer "which days of week does this event occur on?" by just reading the array, instead of deciphering the string).

For **exceptions** in JSON, we have a few choices: - **List of exclusions:** We could have an array of dates for exceptions, like `"excludeDates": ["2025-12-25", "2026-01-01"]`. - **Overrides:** We could have a dictionary of specific date -> override-event info. For example:

```
"overrides": {
    "2025-12-25": { "cancelled": true },
    "2025-11-01": { "startTime": "10:00", "summary": "Special time this day" }
}
```

This way, if one instance is cancelled or rescheduled, it's captured in one place under the main event. This is something JSCalendar does – it has a concept of "recurrenceOverrides" where each overridden instance is a patch of changes to the main event. This avoids creating full duplicate event objects for each exception. - Or we might simply break the event series into separate normal events if exceptions are complex, but that loses the connection that they were meant to be one series.

AI and Recurrence: Recurrence rules are inherently tricky for humans, but a structured format helps. An AI interacting with the calendar might need to generate occurrences or answer questions like "Is there a meeting on July 4?" For iCalendar, the AI would have to apply the RRULE logic or have a library to do so. For a JSON format, if we choose, we could even include an explicit list of next few occurrences or something (though that can get out of date). More realistically, the AI would still compute occurrences, but doing so from a clear JSON rule is easier than from a compact string.

Edge Cases and Legacy Behavior: In iCalendar, some combinations can get complex: - Recurrence rules can include dates and time together with timezones – which means the expansion needs to consider daylight savings changes (e.g., a weekly 9am meeting will shift in UTC time when DST changes). iCalendar handles that by tying the expansion to the local timezone recurrence set; a JSON format would need to clarify the same (likely the recurrence is considered in the local time of the event's start, which is how most systems behave). - One could consider simplifying by disallowing certain convoluted rules. For instance, iCalendar lets you have multiple RRULEs or mix RRULE with RDATE. Many implementations have bugs if you have more than one RRULE. A simpler approach: allow only one recurrence rule per event, and use separate events if you need a totally different pattern. Or disallow the exotic BYSETPOS or complex by-component combinations in a minimal spec (they are rarely used outside of niche cases).

However, since you said *all aspects (including recurrence) interest you*, a better approach is to include robust recurrence but make it more JSON-friendly. That likely means yes to supporting rules like "every X weeks on these days" or "every year on the third Thursday of November" because those are real use cases (think Thanksgiving). JSON can capture those clearly with the right fields.

Non-Gregorian in JSON: If we did want to allow non-Gregorian patterns, we could incorporate something like:

```
"recurrence": {
   "frequency": "yearly",
   "calendar": "hebrew",
   "month": 1,
    "dayOfMonth": 10
  }
```

for “10th day of 1st month of Hebrew calendar every year”. And perhaps a skip rule if needed. But again, this might be something to add later or design as an extension to keep the base spec simpler.

Citing the RFCs for Recurrence: It might be useful to note: - The iCalendar spec emphasizes the importance of unambiguous date-time with time zone for recurrence computations [34]. That’s why VTIMEZONE exists. So in JSON, we ensure we have TZIDs or offset-aware times for recurrence start. - RFC 5545 removed the old EXRULE and relies on EXDATE and multiple VEVENT exceptions as described [35]. We can do better in JSON as noted. - RFC 7529 (non-Gregorian) basically solved a specific legacy problem (CALSCALE) by a modern tweak to RRULE [21] [4]. This suggests that *if* you foresee your format being used globally, eventually supporting other calendar systems is a good idea, but it can be done in a way that doesn’t complicate everything else (only those events that need it would have an extra field indicating the calendar system).

In summary, **recurrence rules can be made much more accessible in a JSON format**. We trade a bit of verbosity for clarity. A minimal JSON calendar could initially support basic recurrence patterns (daily, weekly, monthly by date or day, yearly) which cover 99% of cases, and omit extremely complex rules. This covers personal calendaring needs (like “recur every weekday” or “recur on the 1st of each month”). If needed, more complexity can be added later in a controlled way. Crucially, the JSON approach frees us from the straitjacket of the RRULE string and the juggling of multiple event components for exceptions – these can be handled with more direct representations.

Synchronization and Scheduling Considerations

Even if we focus on a personal calendar (which often implies events you create for yourself), real-life use involves sharing events (invitations) and syncing across devices. Let’s look at how iCalendar handles these and how a new format might.

iTIP Scheduling (Invitations): The iTIP protocol (RFC 5546) defines how to use iCalendar data for scheduling between calendar users [15]. It’s essentially a set of **methods** and rules for updating events: - An **organizer** sends a meeting request (METHOD:REQUEST) with the event details to all attendees [36] [37]. - Attendees reply with METHOD:REPLY, echoing the event UID and indicating their `ATTENDEE:...PARTSTAT=ACCEPTED` (or TENTATIVE/DECLINED) [38] [39]. - If the organizer updates the event (time change, etc.), they send another REQUEST with a higher SEQUENCE number. Attendees then know to replace the old one. - If the organizer cancels, they send METHOD:CANCEL (which is a stripped-down event with STATUS:CANCELLED typically). - There are also less-used methods: PUBLISH (send out an event with no specific attendees, e.g., a public calendar posting), ADD (to add an instance to a recurrence set), CANCEL for just one instance, REFRESH (ask for latest info), COUNTER (attendee proposes a new time), DECLINECOUNTER (organizer rejects the counter-proposal) [40] [41].

All these travel via some transport. Often it’s email (then it’s called iMIP). The .ics file is attached or inlined. Systems parse them and update user calendars accordingly. This works but has many points of failure (spam filters, different interpretations, etc.), and it’s user-unfriendly if exposed (which is why most UIs hide the .ics and just show accept/decline buttons).

Synchronization (CalDAV and others): Sync is about keeping your calendar consistent on multiple devices and with a server: - **CalDAV (RFC 4791):** This is widely used in Apple, Google (they support a version of CalDAV for syncing), and other clients. It treats the server as a store of iCalendar objects. Each event is a separate resource (usually one .ics per event UID). Clients use HTTP methods to fetch events (GET), add events (PUT new .ics), change events (PUT updated .ics), or delete (DELETE). CalDAV introduced mechanisms like `sync-collection` reports to efficiently sync changes, and uses ETags to detect modifications. It's quite complex under the hood (as it extends WebDAV), but from the client perspective it's a way to retrieve batches of events or get incremental updates. CalDAV's scheduling extension (RFC 6638) allows the server to handle sending out iTIP invitations automatically when you create events with attendees – effectively moving some email exchange to the server-to-server realm. RFC 7953 updated CalDAV to handle the new VAVAILABILITY component as well [42]. - **JMAP for Calendars:** A newer JSON-based API (still being adopted) where clients communicate with a server via JSON over HTTP. The data model in JMAP is JSCalendar objects rather than iCalendar. This protocol is designed to be mobile-friendly (efficient batching, push, etc.). The idea here is, if you have a JSON format already, syncing can be done in a much simpler, cleaner way than parsing iCalendar over WebDAV.

- **Subscriptions:** There's also the concept of subscribing to a read-only calendar feed (like holidays or a colleague's public calendar). This is often just done by fetching an .ics URL periodically. If we had a JSON format widely used, one could similarly fetch a `.json` calendar feed.

For a **new JSON calendar format**, considering sync: - If you don't worry about existing tooling, you can design an API from scratch. Perhaps each event is a JSON resource accessible via REST. Or even simpler, if it's just the user's own devices, maybe the calendar could live in a cloud storage (like a single JSON file or a database). But likely a proper API (like a mini CalDAV) is needed if multi-client sync is in scope. - One possibility: Provide a conversion layer (so that externally it can talk iCalendar for compatibility, but internally use JSON). But since you said it's time for a new way and not to worry about existing, you might not prioritize that. - If multiple users are to schedule together, either everyone uses this new JSON format and exchange invites in JSON form (maybe via a web service or email with JSON attachment), or there's a gateway to iCalendar for talking to legacy systems.

Invites in JSON form: Instead of emailing an .ics, one could email a .json (or a link to a shared event). However, email programs won't understand that yet. Another approach is to handle invitations through a server: e.g., all participants are on a shared server or send each other events through a network request. The format itself simply needs to hold the necessary data (list of attendees and their responses). The actual process of inviting (how the event gets from one person's calendar to another's) could leverage existing channels like email or messaging apps but with the new format attached.

Attendee Status Tracking: The JSON event could include an `attendees` array where each attendee object has a `status`. When someone accepts, their status changes to "accepted" in the organizer's copy (and in everyone's copy, ideally). Achieving that requires a mechanism (the attendee sends a reply which updates the organizer's copy). If using a central server, it's straightforward (the attendee's client just updates a status field via an API call). Without a server, it's trickier – you'd have to send the JSON back to the organizer somehow. This starts to reinvent what iTIP does, but one could think of a simpler version: maybe just send the whole event JSON with a flag updated, or send a tiny JSON snippet indicating acceptance referencing the event ID.

Free/Busy via Availability: For personal use, free/busy might not be explicitly exchanged, but if you share your calendar with someone, the JSON data itself (or a filtered version) could be shared. Alternatively, one could have an endpoint like `/freebusy?range=...` that computes from the events. Since JSON format would make it easy for any service to parse events, that's not a problem.

Considering AI and "smart" scheduling: If an AI assistant is managing your calendar, it could use the JSON format to do things like: - Check your availability by loading your events and seeing open slots. - Propose new event times and easily adjust the JSON. - Communicate with another person's AI to compare calendars (if both use a similar format or the AI can convert). - The simplicity of the format could allow it to identify conflicts or travel time gaps more straightforwardly.

Versioning and Conflict Resolution: Sync often needs to handle conflicts (two devices edit the same event differently). iCalendar's SEQUENCE and LAST-MODIFIED help a bit, but real conflict resolution is usually left to servers or clients (who may prompt the user). In a new format, you'd likely incorporate a lastModified timestamp and maybe a unique ID per event revision (like CalDAV's ETag). This isn't so much a format issue as a sync protocol issue, but the format should include at least an `updated` timestamp so clients/AI can tell which copy is newer.

Opinion – How far to go with scheduling: Since you're focusing on personal calendar, one approach is to postpone the full scheduling feature set. That is, define the event format (with attendees fields) but not completely specify how invites are transported. That could be handled by a higher-level system. In contrast, iTIP was tightly linked to iCalendar. If we decouple them, your JSON format can be used for events regardless of how invites are sent (could be via a specific API or even converted to iCalendar for emailing to someone not on the system). This separation of concerns might make your format simpler. Essentially: *format first, protocol later*. And when designing the protocol, one could opt for something modern (maybe RESTful calls or pub/sub).

Interoperability vs Innovation: My opinion is that a new format should not be bogged down by old protocols; however, if it ignores them entirely, it risks isolation. The good news is, translating between JSON Calendar and iCalendar is possible (like how JSCalendar is designed to round-trip from iCalendar). That means, if absolutely needed, a bridge can allow legacy systems to interact. But focusing on innovation, you can assume a fresh ecosystem: e.g., a cloud service where calendars are stored as JSON documents, clients use a simple API, and AI agents have full read/write via that same API.

Example – a simple sync scenario: Imagine a minimal implementation: a calendar is just a JSON file stored in a cloud drive (like Dropbox, etc.). All your devices edit that file (maybe using some lock or last-write-wins if conflicts). That's as simple as it gets (no specialized protocol at all). It's not efficient for big calendars, but for personal use maybe fine. An AI could also read that file if given access. However, this doesn't scale well or handle multi-user invites elegantly. So likely, a more structured approach akin to JMAP/CalDAV is needed if multiple users are interacting.

Security and Permissions: In a new system, you'd consider how to share a calendar or event with others securely (maybe via access control on a server or by invitation links). iCalendar's CLASS:PRIVATE doesn't enforce anything; it's up to the server or client to hide details. In a modern context, one might have specific sharing settings rather than marking events private within the data.

In conclusion, **synchronization and scheduling** in the context of a JSON calendar format could be dramatically simplified by using modern web paradigms: - Use HTTP/JSON APIs for sync (no need for XML WebDAV intricacies). - Perhaps leverage push notifications or event streams for instant updates. - Exchange invitations by direct client-to-client communication through a service (rather than email attachments), or by simply adding the event on a server where attendees can see it. - The data format (JSON) is easier to integrate into these workflows than iCalendar, which often required transformation to/from text.

While some of these choices go beyond the format itself, it's important the format can support the necessary info (like attendees, their RSVP status, event identifiers, and modification timestamps). The leaner the format, the easier it is to use in any new protocol or AI agent logic. This complements your goal: a **minimal, AI- & human-readable calendar data format** that isn't weighed down by legacy.

Designing a Minimal, AI-Friendly JSON Calendar Format

Bringing it all together, what would an ideal JSON Calendar format look like, based on what we've learned?

Core Design Principles: 1. **JSON Native:** Use JSON objects/arrays to represent the structure (one calendar containing events, events containing subcomponents like attendees or alarms). No more pseudo-code strings like RRULE or semicolon-separated parameters – everything becomes explicit JSON fields. This inherently makes it more readable and writable by both humans and AI, since JSON is widely understood. 2. **Focus on Events:** Limit the scope to events (and possibly availability/free-busy as a separate construct). By not including tasks or journal entries at all, we avoid a lot of complexity. As noted, iCalendar's inclusion of VTODO and VJOURNAL is legacy for our purposes [19]. Those who need task management can use dedicated task formats or a future extension, but your calendar format can remain purely event-oriented, which also aligns with how many users think of "calendar". 3. **Human-Readable Field Names:** Use descriptive keys like `"title"`, `"start"`, `"attendees"`, etc., instead of cryptic codes. This means someone looking at raw data can guess the meaning, and an AI doesn't need a large table of property name mappings. (Even for recurring rules, keys like `"frequency"` and `"interval"` are understandable). 4. **Minimal necessary fields:** An event might only require an `id`, `start`, and `title` to be valid (maybe end, if not a default duration). Everything else can be optional. This is similar to iCalendar where UID, DTSTART, and a few things are required, but we can simplify even more (for instance, if an event has no UID, perhaps the storing system could assign one, but generally we do want a stable id). 5. **Drop or simplify rarely used features:** As we enumerated in the legacy section – no need for things like `SEQUENCE` (the protocol or system can handle versioning), no need for `CLASS` if we handle privacy differently, etc. If something is needed only in advanced enterprise scenarios (like delegation or complex recurrence rules), consider leaving it out of the initial spec. You want a *minimal* viable format that covers personal calendaring well. 6. **Extensibility:** Even though we strip out a lot of legacy, we should allow adding new fields in the future in a way that doesn't break things. JSON is good here: clients can ignore fields they don't understand. We could also namespace or have an `"extensions"` container for custom stuff. iCalendar allowed "X-" properties for extension [12], and a JSON format could allow vendor-specific fields or extension keys without disrupting core data (especially if using a schema/validation approach where unknown fields are tolerated). 7. **Internationalization and Unicode:** JSON inherently supports Unicode, so we can effortlessly include non-ASCII text (event names in any language, emoji, etc.) without the special encoding rules that iCalendar needed (no more `CHARSET` or `ENCODING` issues). This makes it friendly for global use and for AI that might need to handle text analysis on event content. 8. **Time Zones and Date Formats:** Embrace ISO 8601 date/time strings (with timezone info if needed) or a clear structured representation. This avoids confusion of floating vs UTC times. Possibly default to ISO strings like

`"2025-06-20T12:45:00+02:00"` (which includes offset) or use `"timeZone": "Europe/Stockholm"` as a separate field. The key is consistency and clarity. An AI will recognize an ISO timestamp easily. 9. **Recurrence Clarity:** Represent recurrence rules as sub-objects as discussed, or at least as a well-defined array of rules. Avoid requiring the client to interpret natural language or complex strings for something the computer can handle. We learned from iCalendar's evolution that supporting recurrence is important, but we can present it in a more straightforward way.

What to Keep (from iCalendar): Not everything in iCalendar is "legacy" – a lot of it is essential: - The concept of a **UID** for each event (so it can be referenced for updates) is important [23]. Our JSON events should have a stable `id` field (preferably unique across space and time, like a UUID or some composite of event info). - **Recurrences** and **exceptions** – a calendar without repeats would be severely limited. We definitely include recurrence, just in a simplified form. - **Attendees and organizer** info – even for personal calendar, many events involve others, so we need to represent invitees. We keep that, just in a nicer way. - **Alarms/reminders** – people use calendar reminders heavily. We will include a way to specify alarms. Maybe just a basic implementation (one default alert 10 min before), but ideally the format can do multiple alarms and different types (notification, email, etc.), much like VALARM but simpler. For example, a boolean `{"useDefaultReminder": true}` or an array of alarm objects as earlier discussed. - **Basic event fields** (title, description, location, start/end, etc.) – obviously keep those, as they are the heart of an event. - **Linking** – linking to external info (like a `url` field for an event link) is useful. The RFC 9253 idea of LINK could be as simple as having a list of related links or an attachments list in JSON.

AI Perspective: One reason to have an AI-readable format is to enable smart assistance: - An AI could merge information (like read an email and create an event from it, directly populating the JSON fields). - It could reason about your schedule (e.g., "When is my next free 2-hour slot to exercise?" by scanning event times). - It might also integrate with other AI tools – for example, link a meeting event with an AI that can summarize meeting notes afterwards (the AI could drop a summary into the event's description or a structured-data field). - If events have a structured location, an AI can easily fetch weather or travel time for that location and update the event or alert you. - If the data is cleanly structured, training models or writing rules on it is much easier. By contrast, learning from raw .ics files is not something you'd expect an ML model to do due to their format quirks.

Your Thinking and Feasibility: You propose not worrying about existing tools, which is bold but reasonable for innovation. In the past, backward compatibility often constrained calendar format improvements. By sidestepping that, you can design for the future. However, one opinion: be aware of the **network effect** of calendars. If your format is too isolated, it might be hard to use it in real life where others might invite you to meetings via Outlook or Google. But perhaps you imagine a self-contained ecosystem or at least writing converters. Given that JSCalendar is already an IETF standard (albeit new), aligning somewhat with it might actually help your format gain traction or interoperate with JMAP servers that speak JSCalendar (since they share goals). That said, you can definitely trim even more (JSCalendar, for instance, still includes VTODO equivalent and other complexities – you can drop those as you wish).

In my opinion, the time is ripe for a **simplified calendar format**. People have been layering APIs over iCalendar to hide its complexity (e.g., Google's JSON API for Calendar essentially converts ICS to JSON when you use their service). Rather than each vendor inventing their JSON schema, a unified open format (like what you're aiming for) could be very useful. It could live alongside iCalendar for a while, but if done well, it might replace it in new applications (especially those involving AI or web integrations).

Summing Up Requirements for JSON Calendar: - Represent events with clear fields (id, title, times, etc.). - Support recurring events with structured rules. - Allow cancellations or modifications of individual instances of a series. - Include participants info for events (at least basic invitation tracking). - Include alarm/reminder capability. - Possibly include a way to indicate general availability (or that could be separate from the event format). - Must be serializable to a file or over an API easily (JSON is by nature). - Be extensible for future additions (maybe define how unknown properties are handled). - Should be fairly easy to convert to iCalendar (for interoperability gateway) – this is not a must, but if you use similar concepts, conversion is straightforward, which is a nice safety net.

By learning from iCalendar's RFCs, we can avoid repeating historical mistakes: - Don't make it overly verbose or angle-bracket heavy (hence not XML). - Don't lock into a rigid schema that can't evolve (hence use JSON which is flexible). - Address known use cases like online meetings, which iCalendar had to patch in later (we can bake that in from start, e.g., a field for video link). - Keep the format independent of transport, but design it such that it works well with modern transports (HTTP, etc., rather than email). iCalendar was designed for email and files; we target web and realtime uses.

Finally, **validating and parsing** this JSON will be simpler. Developers could use JSON Schema to validate a calendar file, something much harder to do with iCalendar's ABNF-based validation. This again means fewer bugs and more reliable interop when different tools use the format.

Opinion: Your idea of a new JSON format, focusing on personal calendar events and trimming legacy, is not only reasonable but aligns with where the industry is slowly heading. It's like a fresh start that can incorporate 20+ years of experience. My only caution is to ensure not to throw away things people actually find useful. For example, some "legacy" iCalendar features like tasks are indeed separate concerns, so dropping them is fine (it doesn't mean tasks go away, they just wouldn't be calendar events). Journals are rarely used, so no loss. But something like free/busy or availability might seem old-school, yet the underlying need (to know when you're free) is very much alive – you just might solve it differently (maybe your AI handles that rather than a static construct).

Also, consider how to handle **multiple calendars**. In iCalendar, a .ics file can contain many events (and typically a user might have multiple calendars like "Work" and "Personal"). In a JSON system, you might represent each calendar as a collection (array) of event objects, possibly with its own metadata (like name, color). RFC 7986's calendar properties (NAME, COLOR, DESCRIPTION for the calendar itself) would translate into fields for a calendar object [43]. So your data model might have a top-level concept of Calendar containing events. This helps AI or clients differentiate different calendars (which might have different sharing settings or purposes).

To wrap up, the modern JSON calendar format would take the rich functionality of iCalendar (as refined by all those RFC updates) but present it in a clean JSON structure, omitting the baggage that isn't needed for straightforward calendaring. This makes it both **human-readable** (someone could inspect or manually edit their calendar JSON if needed, much like one can with simple data formats) and **AI-friendly** (any AI/ML process can parse JSON easily and doesn't have to include an iCalendar parser or knowledge of odd quirks). It's a logical evolution akin to how JSON replaced XML in many domains for being lighter and easier.

Conclusion

The calendaring landscape has been shaped by the iCalendar format's longevity and extensive features. By researching the key RFCs (RFC 5545 and its updates: 5546, 6868, 7529, 7953, 7986, 9073, 9074, 9253), we see a clear picture of both the strengths and the accumulated quirks of the existing standard. The **current state** is that we have a very capable but complex text format that has grown to accommodate scheduling, rich event data, and broad compatibility. This served us well to ensure any two calendar systems can talk to each other, but it also means carrying forward decisions from the '90s computing environment (line-oriented text, email-based exchange, etc.).

In designing a **new JSON Calendar format**, we have the chance to shed the **legacy features** that aren't relevant to most users' needs – such as to-do and journal components, and various outdated encoding issues – and to streamline event representation, recurrence, and synchronization using modern conventions. We'd keep the core concepts that people expect (events, repeats, invites, reminders), yet implement them in a much cleaner way. The analysis of these RFCs suggests that nearly every "pain point" of iCalendar was later addressed by an extension, which provides guidance on what to include (or exclude) in a reimagined format: - We ensure events can handle all the common needs (multi-attendee meetings, online meeting links, attachments, etc.) – the *what* remains the same, but *how* we represent it becomes simpler. - We remove rarely-used or redundant constructs (like those legacy to-do specific fields, or weird scheduling method edge cases) – trimming the fat for a leaner spec. - We make the data structure intuitive. This benefits not only developers but also AI systems that will increasingly manage and interpret our schedules. A JSON format means such systems can leverage vast ecosystems of JSON tools and don't have to be calendar experts to do basic things.

In my opinion, your thinking is on the right track. The evolution of iCalendar itself (especially with JSON-based jCal and the new JSCalendar) validates the idea that JSON is the future for calendar data. By focusing on personal calendars and not being tied to backward compatibility, you can create a format that is both **minimal and powerful** – minimal in complexity, powerful in expressing what needs to be expressed. The end result could be a format that is easy to read (for a human glancing at raw data or configuration), easy to generate or manipulate (for an AI or script adjusting your schedule), and easy to extend (for future needs we haven't thought of yet, without requiring a new RFC each time).

Adopting such a new format would require some ecosystem support, but if it clearly outshines the old in convenience, it could start as a translation layer or for niche use (like personal AI assistants) and gradually prove its worth. Just as JSON has taken over REST APIs and config files because of its simplicity, a JSON calendar could eventually replace .ics in everyday usage – especially as our tools become smarter. It's an exciting direction, and the lessons from the past standards will help ensure the new design avoids their pitfalls while preserving their capabilities.

Sources:

- *RFC 5545* – Internet Calendaring and Scheduling Core Object Specification (iCalendar) [1] [44]
- *RFC 5546* – iCalendar Transport-Independent Interoperability Protocol (iTIP) [15] [16]
- *RFC 6868* – Parameter Value Encoding for iCalendar and vCard (allowing special characters in properties) [45] [46]
- *RFC 7529* – Non-Gregorian Recurrence Rules for iCalendar (RSCALE extension) [2] [4]
- *RFC 7953* – Calendar Availability (VAVAILABILITY component for free/busy times) [13]

- *RFC 7986* – New Properties for iCalendar (e.g., NAME, IMAGE, CONFERENCE, COLOR) [5] [6]
- *RFC 9073* – Event Publishing Extensions to iCalendar (structured data, VLOCATION/VPARTICIPANT, etc.) [7]
- *RFC 9074* – “VALARM” Extensions for iCalendar (acknowledgments, snooze, alarm UID, proximity triggers) [8] [9]
- *RFC 9253* – Support for iCalendar Relationships (LINK, CONCEPT, new RELATED-TO types) [14]
- *RFC 7265* – jCal: The JSON Format for iCalendar (mapping .ics to JSON) [47] [18]
- *RFC 8984* – JSCalendar: A JSON Representation of Calendar Data (a modern JSON calendar format aligning with JMAP) [48] [49]

---

[1] [19] [22] [34] [35] [44] RFC 5545 - Internet Calendaring and Scheduling Core Object Specification (iCalendar)
https://datatracker.ietf.org/doc/html/rfc5545

[2] Information on RFC 7529 » RFC Editor
https://www.rfc-editor.org/info/rfc7529

[3] [4] [20] [21] [32] [33] www.rfc-editor.org
https://www.rfc-editor.org/rfc/rfc7529.txt

[5] [6] [23] [27] [28] [29] [43] iCalendar.org - New Properties for iCalendar (RFC 7986)
https://icalendar.org/RFC-Specifications/iCalendar-RFC-7986/

[7] [30] [31] RFC 9073: Event Publishing Extensions to iCalendar
https://www.rfc-editor.org/rfc/rfc9073.html

[8] [9] [10] [11] [12] [24] [25] [26] RFC 9074: "VALARM" Extensions for iCalendar
https://www.rfc-editor.org/rfc/rfc9074.html

[13] [42] www.rfc-editor.org
https://www.rfc-editor.org/rfc/rfc7953.html

[14] RFC 9253: Support for iCalendar Relationships
https://www.rfc-editor.org/rfc/rfc9253.html

[15] [16] [36] [37] [38] [39] [40] [41] www.rfc-editor.org
https://www.rfc-editor.org/rfc/rfc5546.html

[17] [47] iCalendar - Wikipedia
https://en.wikipedia.org/wiki/ICalendar

[18] Standards - Introduction - CalConnect
https://devguide.calconnect.org/Appendix/Standards/

[45] [46] www.rfc-editor.org
https://www.rfc-editor.org/rfc/rfc6868.html

[48] [49] RFC Search Detail
https://www.rfc-editor.org/search/rfc_search_detail.php?sortkey=Date&sorting=DESC&page=All&title=iCalendar