Chat Server Administration Guide

Deployment guidelines for async and regular chat

12/16/2025

Deployment guidelines for async and regular chat

The page provides some important guidelines regarding regular (traditional) and async (asynchronous) chat deployment.

- Comparison of regular vs. async chat mode
- Comparison of short polling (REST-API-based) vs. CometD-based chat
- Sizing guidelines based on comprehensive stress testing
- Async chat workflow recommendations
- How disconnects and idle timeouts work

Regular vs. async chat mode

In general, both async and regular chats are processed the same way by all components. However, async chat provides additional capabilities that require a bit more planning and workflow implementation.

	Regular Chat	**Async Chat**
Duration of single conversation	Lasts only minutes or dozens of minutes.	Could potentially last for days or even weeks.
Agent handling	An agent can: • Accept the chat session • Transfer the chat session • Stop the chat session	In addition to the agent handling found in regular chat, an agent can also: • Place the chat session on-hold • Resume the chat conversation at a later time by re-activating the interaction from the workbin
Mobile oriented	Can be implemented, but not suited for lengthy conversations.	Suitable for mobile applications as it permits lengthy conversations with periods of inactivity.
Workflow	Mostly used for routing purposes (in other words, selecting the best available agent).	Additionally, must handle re-activation of interactions placed on hold after a qualified event occurs (for instance, a new incoming message from a customer, or an async idle timeout expiration).

	Regular Chat	Async Chat
Performance implications	Must be sized to conduct a certain number of active chat sessions.	Must take into the account the presence of a large number of chat sessions, most of which are expected to be in a dormant state. Please see below about sizing guidelines.

Short polling vs. CometD-based chat

End-user (customer-oriented) web or mobile chat applications must communicate with Genesys Mobile Engagement (GMS) through two alternative APIs:

- Short polling (REST API) - For this API, the chat application is required to send frequent (usually every other 3 seconds) polling requests to keep the chat session transcript updated.
- CometD-based - This API can utilize either WebSockets or long-polling. This provides chat session transcript updates more promptly, on the customer side.

While the CometD approach naturally appears more efficient (as it also reduces the overall load onto the system by eliminating unproductive API calls), the table below provides a comparison of different aspects which should be taken into account when selecting the best approach for your deployment and implementation:

	Short polling (REST API)	CometD-based
Performance	Consumes more CPU and traffic resources as it produce a lot of unproductive API calls (according to research, on average 98.5% of polls are wasted). So, if a message is expected to be posted into the chat session every 30 seconds, an extra 10 unproductive API requests must be processed during this time by the GMS and Chat Server components. Basically, this means that the load on these components are measured on how many concurrent chat sessions an instance can hold, independently of the scenario density (see Sizing Guide, Setting Load Limits, and Health Monitoring for more information).	CPU and traffic resources are used mostly for the productive load.
Connections	Each API call is executed on a separate connection, which is closed immediately after receiving an HTTP response. The number of concurrent connections (between GMS and	Number of concurrent connections is similar to the amount of concurrent chat sessions. Also, GMS imposes a limitation of only one CometD connection every chat session.

	Short polling (REST API)	CometD-based
	Chat Server) depend on the number of concurrent chat sessions, divided by the short polling interval (usually 3 seconds).	
Complexity	Simple to implement and troubleshoot (as it is based on pure REST API).	Troubleshooting requires the knowledge of CometD protocol functionality.
Client library	There are numerous stable versions of HTTP REST libraries.	CometD client library is required, which increases the complexity of the chat web application.
Timeouts (in Chat Server)	The flex-disconnect-timeout configuration option is used to disconnect a chat participant who has not sent any API requests after a specified amount of time.	The flex-push-timeout configuration option is used to disconnect a chat participant who is not confirmed by GMS as alive after a specified amount of time.

Sizing recommendations

On the high level, sizing guidelines depend on various factors:

- Short polling (GMS Chat API Version 2) vs. CometD (GMS Chat API Version 2 with CometD) mode:
  - Short polling produces a constant "background" (or, "noise") load onto GMS and Chat Server, thereby consuming much more CPU and network resources. Also, it is important to appropriately tune the operational system Transmission Control Protocol (TCP) parameters to minimize the `TIME_WAIT` state duration, as each short polling request leads to the opening and closing of the TCP connection.
  - The CometD approach requires you to keep a long-living connection to GMS for each chat session. It should be noted that some load balancing solutions do not handle long-living connections properly and might result in the closure of an inactive connection.
- In chat async mode, dormant vs. active sessions:
  - Active chat sessions usually constitute only a fraction of all ongoing async chat sessions. The number of such chat sessions should be around the number of active chat agents, multiplied by the capacity of agents (or, how many parallel chat session an agent can work on). These chat sessions consume almost all assigned resources (first of all CPU).
  - Dormant chat sessions are those which do not have an active agent (or bot) in the chat session. So, for example, in short polling mode the customer-facing application must minimize the resource consumption by reducing (or completely eliminating) the periodic short polling requests.
- Scenario density: Using a CometD approach, the load scales linearly with the number of messages sent from chat participants during a certain time period. Using the short polling approach, the load is mostly dependent on the “noise“ load, since polling requests take up most of the packets being processed.
- In High Availability (HA), UCS vs. Cassandra:
  - UCS-based HA option requires less deployment and maintenance efforts, and guarantees the presence of the latest transcript version for ongoing chat sessions in the UCS database (DB).

However, with large deployments, UCS and UCS DB might be overloaded with intermediate transcript updates which are generated by Chat Server after each chat session message.

- Cassandra allows you to off-load UCS from an unnecessary load. However, in the case of an unplanned Chat Server switch-over during the ongoing chat session, the chat transcript can never be propagated into the UCS record if the chat session cannot be restored on another Chat Server instance (in other words, when it coincides with a customer-facing chat application failure or closure).

Important

For async chat, especially in short polling mode, a customer-facing web or mobile chat application must noticeably reduce the frequency of short polling requests when it detects that a session was placed on hold (in other words, when the agent leaves chat session.

Performance benchmarks

The following benchmarks were produced on:

- Hardware with "Intel Xeon E7-8880L 2 GHz" with a single instance of Chat Server which consumed in average one CPU core. At the maximum possible load, Chat Server can consume no more than 2 CPU cores since all chat-processing operations are executed in a single thread (similar to `Node.JS`) and only auxiliary activity operations are executed by a few other working threads.
- Two instances of GMS, each consuming approximately one CPU core.

The average length of a chat session was around 35 seconds (with 3 messages from a customer and 3 messages from an agent), which is a very dense scenario. Each cell in the table contains the total number of concurrent chat sessions (and active vs. dormant ratio).

Mode	Active to dormant sessions ratio		
All active	**1:10**	**1:50**	
Short polling mode	1000	8000 (800 / 7200)	35000 (700 / 34300)
CometD-based mode	1500	11000 (1000 / 10000)	39000 (900 / 38100)

Important

These are performance load test benchmarks; these numbers are not expected in a real-life scenario.

Async chat workflow recommendations

For async chat, the workflow (or the set of Universal Routing Server [URS] or Orchestration Server [ORS] strategies) must additionally provide the handling of chat sessions being placed on hold by an agent to the regular chat workflow. The on-hold session can be processed in the follow ways:

- Upon the qualified event (a message or a configured notice) in the chat session from a customer, Chat Server updates a special key-value pair (KVP) in the corresponding interaction, which is handled by Interaction Server. As you see it implemented in the Chat Business Process Sample, the workflow can force the interaction for routing, which routes the interaction to any other agent after several attempts of trying to route it to the last handling agent.
- Alternatively, the workflow can place the interaction back to the last handling agent's workbin, if that last handling agent is not available at the moment. However, in this case, the workflow must implement the "escape" to avoid this interaction being stuck forever, if that last handling agent never comes back to the interaction.
- With a custom desktop, the workflow might not force the interaction to routing at all upon the qualified event. In this case, the agent desktop can directly subscribe to notifications from Interaction Server when the interaction properties are changed in the agent workbin.
- The workflow must ensure that interactions are not stuck when placed on hold. In the Chat Business Process Sample, this is implemented in `async-chat-main-check-view` view of `async-chat-main-queue` with the condition `GCTI_Chat_AsyncCheckAt < _current_time()`, where `GCTI_Chat_AsyncCheckAt` is set by Chat Server to the sum of async-idle-alert and async-idle-close configuration options of Chat Server application.

How disconnects and idle timeouts work

Chat Server configuration options allows you to specify timeouts to control two different functional areas:

- The disconnect of a chat session participant, which leads to the removal of a chat participant from a chat session.
- The absence of an activity from participants in a chat session, which leads first to an alert notification and then the closing of a chat session (if no activity is being produced since the alert was sent).

To describe each functional area, the following definitions must be introduced:

- "Protocol inactivity" means the absence of any protocol requests from a client to Chat Server for a certain period of time. It is used to detect the disconnect of a chat participant. For example, if the client application sends short polling refresh requests it still resets the timeout for protocol inactivity even if it does not carry any useful load. So, the client is considered active on the protocol communication level.
- "Session inactivity" means the absence of qualified events (such as messages) from the chat participants with full visibility in the chat session. For example, if a customer and an agent are not sending messages for a certain period of time, then it is considered as a session inactivity. If, at the same time, the agent communicates with another agent invisibly from a customer (or, consultation call), it does not affect this decision (as this conversation is not fully visible for all chat session participants).

Important

A chat session stays alive in Chat Server until at least one participant is present. As soon as the last participant leaves, Chat Server closes the chat session forever and it cannot be resumed again.

Chat session participant disconnect and removal

In terms of connectivity, chat session participants can be processed differently depending on how the application (representing the participant) communicates with GMS and/or Chat Server:

- An agent (or bot) participant communicates with Chat Server via persistent TCP network connection, thus the disconnect leads to the immediate removal of a participant from a chat session.
- A chat participant, represented as a customer in a chat session (or, "client participant") can communicate with GMS in three different modes. Each mode utilizes different configuration options:
    - **Short polling (REST API)**. In this mode, Chat Server uses flex-disconnect-timeout which defines the maximum amount of time of protocol inactivity. As soon as the timeout expires, Chat Server removes the participant from a chat session. If this is the last participant, Chat Server closes the chat session.
    - **CometD only**. If a customer web application communicates with GMS over CometD, GMS subscribes to unsolicited notifications from Chat Server for this chat participant immediately after the chat session is successfully created by Chat Server. This request forces Chat Server to disable flex-disconnect-timeout for the chat participant and instead uses flex-push-timeout for the periodic querying of GMS to confirm that the participant is still connected over CometD. When GMS sends the confirmation, it tells Chat Server to consider the chat participant alive. As soon as GMS detects the disconnect over CometD, it sends an "unsubscribe" request, which forces Chat Server to enable flex-disconnect-timeout until the new subscribe request is sent by GMS to Chat Server (upon client re-connection over CometD to GMS).
    - **CometD and short polling with subscription for either mobile or custom-http push notification**. This mode operates almost exactly the same way as "CometD only" except that GMS never sends an "unsubscribe" request upon a CometD disconnect to Chat Server. This forces Chat Server to use only flex-push-timeout to ping GMS. In this mode, flex-disconnect-timeout is activated only when a client chat participant is removed from the chat session forcibly by another chat participant (such as an agent or bot).

Important

When Genesys agent desktops (WDE and WWE) receive the event indicating that a client left the chat session (for any reason), the agent desktop automatically sends the request to Chat Sever to close the chat session. A custom desktop can implement this differently if needed, as Chat Server keeps the chat session alive until the last participant leaves the chat session.

Inactivity control and chats session closure

We define chat session inactivity as the absence of a qualified event in a chat session for a certain period of time (defined by timeouts in configuration). A qualified event can be a message, a notice (as defined by async-idle-notices or include-notices), and a participant (the agent) joining or leaving the chat session. Only events with full visibility (in other words, visible to all participants) are taken into account here.

There are two complementary inactivity control configurations in Chat Server:

- Generic chat configuration (applicable both for async and regular chat sessions).
    - It is enabled:
        - If the option `enable` in section [inactivity-control] is set to `true`.
        - When both a customer and an agent (bots are not considered agents in idle control configuration) are present in the chat session. Once the last agent leaves the chat session, Chat Server disables this configuration.
    - After a certain period of inactivity (defined by option timeout-alert), Chat Server sends an alert notification (with text defined in message-alert option).
    - If there is still no activity (for the period defined by option timeout-alert2), Chat Server sends the second alert (with message defined in option message-alert2).
    - If there is still no activity (for the period defined by option timeout-close), Chat Server sends a "close" notification and immediately closes the chat session.
- Async only chat configuration (applicable only for async chat session):
    - Is enabled from the very start of an async chat session. It is activated independently of the presence of an agent in a chat session (in other words, it is activated even if you have only a customer in the chat session).
    - After a certain period of inactivity (defined by option async-idle-alert), Chat Server sends an alert notification (with text defined in the message-alert option).
    - If there is still no activity (for the period defined by option async-idle-close), Chat Server sends a "close" notification and immediately closes the chat session.

Important

Chat Server resets the inactivity period after any qualified event occurs in the chat session. Both inactivity configurations could be activated simultaneously.